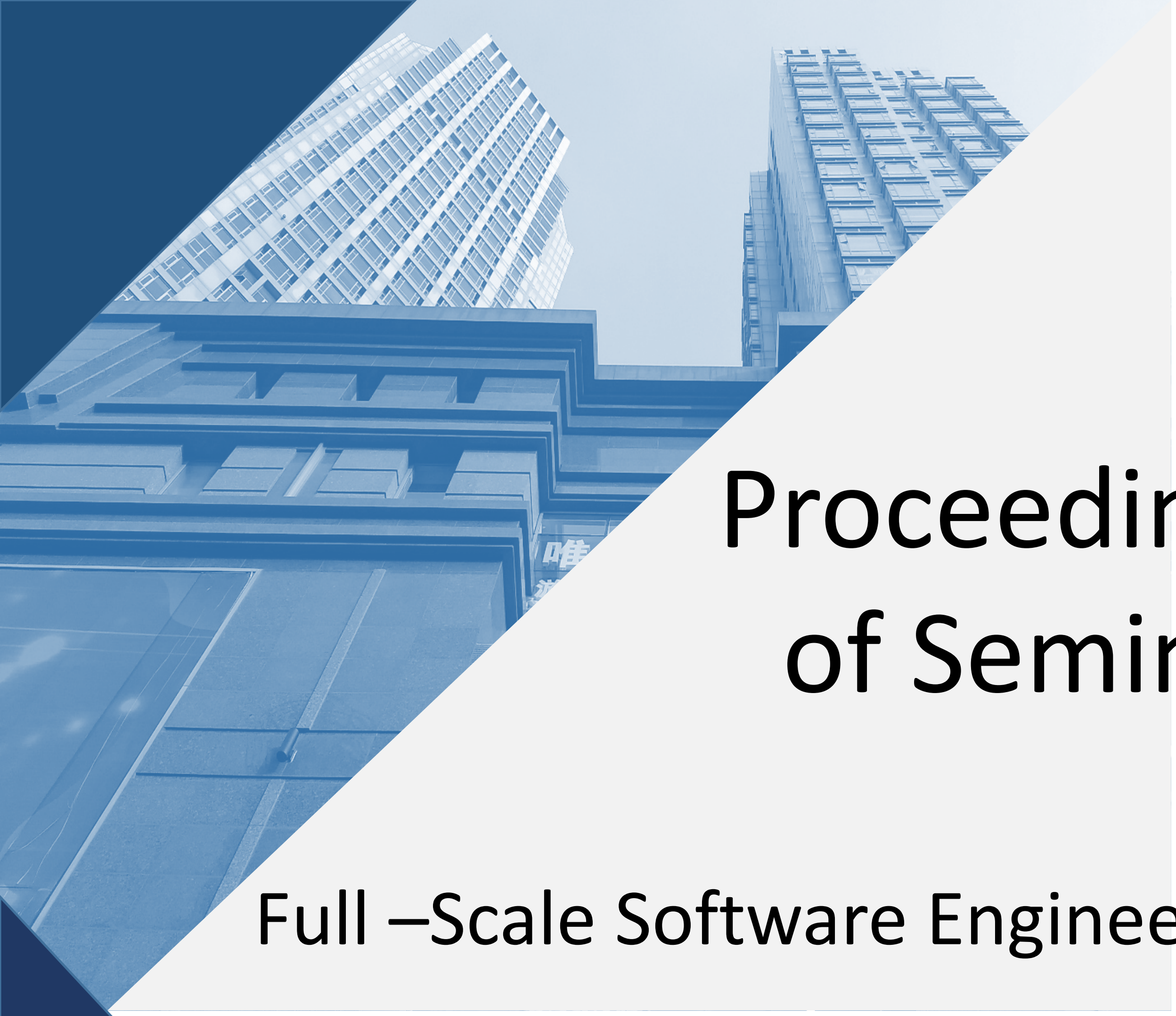


Proceedings of Seminar

Full –Scale Software Engineering

2017

Editors: Horst Lichter
Andreas Steffens
Firdaus Harun
Konrad Fögen
Andrej Dyck
Simon Hacks
Ana Dragomir

Software Construction

RWTH AACHEN UNIVERSITY

Continuous Architecting: Just another buzzword?

Benedikt Holmes
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
benedikt.holmes@rwth-aachen.de

Ana Nicolaescu
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
adragomir@swc.rwth-aachen.de

ABSTRACT

Software architecting is always a core component in software development process. In the last few years a new buzzword 'continuous architecting' came up in the software development community, describing practices avoiding common pitfalls in architecting. While some practices optimize the architecting process, others strive for solving the problem of architecture erosion, whether the actual development process being agile or not. Since most interpretations differ in their understanding of continuous architecting, there is need for a clear definition. This paper gathers definitions of continuous architecting concepts and discusses the novelty of this concept.

Keywords

Continuous Architecting, Agile Architecting, Software Development

1. INTRODUCTION

Software companies aim to optimize their software development process. Agile practices like Continuous Delivery had a huge impact on software developers in the last years, improving their release cycles and responsiveness [5]. These modern development practices also have an impact on architecting practices [4]. Since architecting is a significant element of the development process, which has not yet been optimized successfully [5], it implies the need for practices addressing architectural challenges. Modern approaches that cope with architecting issues have been presented under the name of continuous architecting in the last 2 years.
Architecture describes the "fundamental concepts or properties of a [software] system" [1], while Architecture Description is everything used to express the architecture in any desired form, commonly in form of documents [1]. The Architect is considered to be the person or a group, which are also responsible for producing the Architecture Description [2]. Architecting therefore refers to the activities of the architect [2].

The ability to respond to changing requirements with a suitable change in architecture design is one exemplary understanding of continuous architecting: The classic waterfall model begins with system analysis, which outputs system requirements. These requirements are the main input for architecting practices, which then output an architecture description. Nowadays modern development makes it possible for requirements to change freely with agile development practices. But can architecting cope with this agility and how are changes in requirements regarding the architecture handled? Can architecting practices aid in further optimization of agile development concepts?
While continuous architecting can be related to agile development concepts like Continuous Delivery, other interpretations, even in non-agile contexts, are possible, due to the scope of architectural problems. There is no clear definition for continuous architecting yet and therefore this paper reviews four continuous architecting perceptions to answer the following research questions:

1. What exactly is continuous architecting?
2. Is continuous architecting just a new buzzword?

Section 3 will go into detail on selected problems that occur with standard architecting practices. After that, in section 4, different methods to handle these problems are drawn together. These are all presented under the name of continuous architecting. Concurrently the differences and similarities between these methods are discussed to finally extract a definition for continuous architecting and discuss the novelty of this concept in the conclusion.

2. METHODOLOGY

In this section I briefly present the literature search approaches which I used for this paper. I started to search for literature containing the keywords continuous architecting and continuous architecture with the online research tools arXiv, CiteSeerX, and IEEEXplore. The keywords only appeared in papers from 2015 to 2016 as it is a new trend. Since I only found two papers I extended my search to other materials as book or even online blogs, as these are a common medium for trending topics. I also extended my search by evaluating the found papers for useful references on other research papers and new keywords. The new keywords were agile architecting and agile architecture. In total I found seven suitable sources of which only four explicitly mention continuous architecting.
In section 4 these four main perceptions of continuous architecting are presented and analyzed with two aims in mind:

Firstly, a definition of continuous architecting is extracted from each perception. Secondly, similarities and differences between the perceptions and other existing concepts are gathered. At this point it therefore should be announced that the summary of each concept is limited to satisfy the research questions. So only the aspects relevant to the analysis of each of the four perceptions are mentioned. Further detail can be found in the referenced literature.

3. CHALLENGES WITH ARCHITECTING

As already mentioned a different set of problems can be addressed by each continuous architecting concept, due to the scope of the topic. The presented problems are a brief summary of the most common ones, for that solutions are proposed in the next section.
Classic architecting involves a group of architects planning the structure of a software system and letting developers write the code. Eventually some factors cause the finished products architecture to differ from the original architecture description. Main factors are changing requirements, bad or non-existing communication between architects and developers, as well as developers changing parts of architectural decisions on their own [7]. These architectures are defined as eroded architectures. This may still be a significant problem in both agile and non-agile development processes. The amount of refactoring work grows with the delta between architecture description and the real architecture. Having no or an inaccurate view of the current state of architecture makes finding and fixing errors even more difficult. This causes architectural technical debt to accumulate, and increases fixing time.
Another problem is to deliver value in short- and long-term. Short-term value delivery is ensured by agile development and thus fast release cycles. Long-term value in contrast must be provided by a reliable architecture. Every architecture is defined by its non-functional requirements. Long-term quality therefore corresponds to non-functional requirements and how changes in non-functional requirements are handled.
The possibility to make changes to the architecture depends on how tightly the software components are coupled. Therefore a monolith design, meaning the whole software system being one coherent entity, definitely causes problems with agility. Also, architects without deep knowledge in both the business and the technology have low response to trending architectural and/or development practices. This may lead to yet another conflict between architects and developers.

It becomes obvious that an architecture needs to be more than just a pre-development blueprint. Architecting now starts to become a process, of continuously managing and monitoring the current state of the architecture. Also, there is no main problem, but rather a range of problems:

- lack of communication
- architecture erosion (causing architectural technical debt)
- response to changes in (non-functional) requirements
- monolith designs
- an architects lack of knowledge of both the business and the technology

4. PERCEPTIONS OF CONTINUOUS ARCHITECTING

Here, different methods that are labeled as continuous architecting approaches shall be introduced to discuss the novelty of the continuous architecting concept and find a suiting definition. Since every method focuses on different problems and development contexts, depending on what they aim to accomplish, the definitions and practices vary. Perception I introduces a technical software framework to aid designing and refactoring big data architectures presented under the name of "Continuous Architecting of Stream Based Systems" [3]. Perception II is based on a research paper, which conducts a multiple case study on continuous architecting, mostly in large agile companies [7]. It presents an organizational framework to undertake gaps in architecting practices, like communication issues, and architecture management to tackle architecture erosion and ensure short- and long-term responsiveness. Perception III, also labeled as a continuous architecting approach, introduces guidelines for an architectural style that supports modern delivery methods and is suitable for both agile and non-agile development. Perception IV is some kind of special case of the method presented in perception III dealing with a special architecture design namely micro-services.

4.1 Perception I

The first perception of continuous architecting derives from the big-data domain and focusses on aiding architects in designing and maintaining architectures processing big-data.
The vast amount of software components in big-data architectures makes it difficult to tackle architecture erosion and refactor initial architecture designs. However, according to the authors of the research, the main complexity lies in maintaining the effectiveness of big-data architectures. Effectiveness is defined as such: "Effectiveness, in big-data terms, means that the architecture as well as the architecting process [...] are able to support design, deployment, [...] refactoring [...] of architectures continuously and consistently with runtime restrictions imposed by big data development frameworks." [3] Maintaining and quickly evaluating the effectiveness of a big data architecture is accomplished by using a continuous architecting approach. In the context of big-data the understanding of continuous architecting is the "[support of] continuous improvement of big data architectural designs" [3].
Automated support to aid in continuous architecting is given by the technical analysis framework OSTIA [3]. OSTIA focusses on the development framework Apache Storm [3]. The framework provides help by finding design anti-patterns in the analyzed architectures, checking if the architecture is consistent with runtime restrictions of the development framework and finally visualizing big data architectures in an understandable way. Anti-patterns are defined as design patterns that possibly decrease performance or deployability.
Since the framework specifies on one development framework, details on the anti-patterns and architecture analysis method shall not be presented. Although the way the framework works shall be broadly summarized: Through logical code analysis and runtime monitoring of the running architecture information regarding the architecture is gathered in a repository to be easily used by algorithms, that e.g. ana-

lyze anti-patterns.
So concluding, OSTIA helps to apply a continuous architecting method to maintain architecture effectiveness and therefore makes refactoring of big data architectures easier.

Sadly the definition of continuous architecting is not further detailed in this paper, and rather vaguely described as continuous improvement of bgi-data architecture designs. This may be due to this perception being tightly restricted to the big-data domain and to the Storm development framework. Because the papers focuses on this domain and this special framework, it can be assumed that this method is new.
Although one finding concerning 'domain-less' continuous architecting is shortly mentioned. It is advised to create a running architecture as soon as possible, which then can be refactored during development, instead of trying to optimize the architecture in design time. Another problem solved by this method is architecture erosion, since the current architecture is constantly visualized and refactored.

Regarding the research questions it can be said that the given definition of continuous architecting is the continuous improvement of big-data architecture designs. Since no similar approach could be found, the detailed method presented in this perception is assumed to be new.

4.2 Perception II

The next perception of continuous architecting derives from a research study involving large agile software development companies that apply a component based software architecture. The research aims to find gaps in architecture practices and links these to faults in companies organizational structure.
The proposed organizational restructure resulted in overall improvement in risk management, decision management, and communication among architects and developers. Detailed consequences are reduction of architecture erosion and improvement in short- and long-term responsiveness, hence also reduction of architectural technical debt.
The organizational restructure is specified by the conceptual CAFFEA framework [7]. It defines new roles for architects and teams, as well as processes for communication flow between these parties, to close the gaps in architecting practices. A brief summary of the framework shall be given here: On top sits the Chief Architect who "takes high-level decisions" [7]. The Governance Architects coordinate development teams and monitor architecture erosion. One Team Architect is part of each development team and therefore responsible for the architecture implementation in his team. Good communication is assured through regularly meetings of Team Architects with their superior Governance Architect and Governance Architects with the Chief Architect.
Development teams are split into feature-teams and runway-teams. Developing new features in context of short responsiveness is the main purpose for feature teams. Alongside the feature-teams there are several runway-teams, who are utilized to handle the often low prioritized architecture refactoring tasks, hence ensure long-term responsiveness and lower the architectural technical debt.
The hierarchy of architecture roles closes feedback loops between the development teams and the different architect roles, giving developers a better understanding of the architecture and architects a better visualization of the current state of the architecture.

This research paper does also not specify what the authors definition of continuous architecting really is. Nevertheless long-term benefits are due to a reliable architecture. Therefore continuous architecting can also be only roughly described as the following: the support for continuous development and management of software architecture in large software companies, especially developing embedded software; continuity in this case being provided through regularly meetings as mentioned above. The architecture management is improved by applying the said organizational framework. Also, without further introduction the terms continuous architecting and Agile Architecting are used interchangeably. So this yields the question if this approach is actually a new approach.
To prepare a continuous architecting vs Agile Architecting comparison, a definition of Agile Architecture from 2015 by Waterman *et al.* shall be briefly mentioned: "We define agile architecture as an architecture that satisfies the definition of agility by being able to be easily modified in response to changing requirements, is tolerant of change, and is incrementally and iteratively designed - the product of an agile development process" [8]. The understanding of continuous architecting and Agile Architecting so far share some similarities. Firstly, both strive for a dynamic and responsive architecture improvement approach regarding changes in requirements and being based on agile software development. Though the technicalities of the approach may differ between the two concepts. Secondly, both produce similar results, in reducing architecture erosion, hence reducing architectural technical debt. So the detailed approach behind this continuous architecting perception might be new, but the general idea is not.
This understanding does not differ so greatly from perception I and also some similar problems are solved, e.g. architecture erosion. Both see the faults in architecting practices being caused by bad architecture management and both apply an approach to make architecting more dynamic and close feedback loops. In detail the perceptions' approaches do differ a lot, since perception I introduces a technical framework, whereas the second perception presents a conceptual framework. Also, perception I is mostly restricted to aid refactoring only for big data architectures, while this study is actually a study conducted with companies mostly doing embedded software development. The evaluation of the CAFFEA framework also only took place in these companies. But still the idea behind the CAFFEA framework, due to the frameworks conceptual nature, is easier applicable to other software development domains than the methods from perception I.

Back to the research questions: continuous architecting is defined as continuous architecture development and management to augment long-term value delivery, which is overseen by most agile software development practices. Also, this definition has a strong similarity to Agile Architecting as understood by Waterman *et al.* [8].

4.3 Perception III

There is a need for an architectural approach that supports modern rapid delivery methods. The software development cycle refers to a process description of software development. The development cycle specifies distinct phases in development, e.g. development - build - integration - testing - deployment - maintenance. Often, the last steps in the software development life cycle, e.g. QA-Testing and Deployment, are still error-prone and time consuming, because agile development practices mainly focus on optimizing early steps in the life cycle [6]. These would be build and integration steps.
Murat Erder and Pierre Pureur present an architecting concept defined as continuous architecting, which deals with this lack [6].
Their concept does not mainly provide an aiding framework or a process description, but rather an architecting style defined by six principles. A large set of tools is also presented in this book, but they aid to apply the principles. The specific implementation of the practices depends on the context of the company and its product and therefore the tools will not be presented in this paper.

The six architecture principles are the following [6]:

1. Architect products not projects
2. Focus on Quality Attributes
3. Delay design decisions
4. Architect for change
5. Architect for build, test and deploy
6. Model the organization after the architecture design

Principle 1 suggests that cost and effort for architecting should be used more efficiently. Instead of architecting for small projects, a product-centric focus should be applied to architecting, making use of architectural reusability. For similar products, regarding the requirements, one architecture design can support multiple products. Principle 2 emphasizes that architecting decisions should only base on non-functional requirements, since these are the requirements that explain how the system works and restrict the architectural design choices. For this to work, more effort must be spend on specifying non-functional requirements sufficiently. Principle 3 recommends to only make architecting decisions based on clearly defined requirements. Making design decisions in advance for unknown or insufficient detailed requirements, is an unnecessary loss of time and effort. This principle suggests to develop a minimum viable architecture (MVA), an architecture that restricts itself to the known requirements and is extended only when needed. With this approach the architecture is developed incrementally over time, and saves time designing for unknown requirements. Principle 4 explains how to architect for changing and future requirements. At the start of development, only a few non-functional requirements may be known good enough to influence architectural design decisions, as principle 3 proposes to ignore unclear requirements. Through a loose coupling of software components, making them independent of each other, the design stays simple and allows substitution or addition of software components. With applying this principle, as the authors says, one leverages the power of the small [6]. This means that many loosely coupled software components are more powerful in comparison to one monolith-like architecture regarding changes in requirements. An example for a loosely coupled architecture is the micro-service architecture, which will be discussed in more detail in concept IV. Though not all software systems are easy to be modeled as a set of loosely coupled components. However, loose coupling may still be a mechanism to add future software components to a mainly monolithic-like architecture design. Principle 5 is the first principle that relies on agile techniques being applied to the development process. The first four principles can also be applied to non-agile development and delivery processes. In agile development although parts of the software development life cycle are still time consuming. The architecture must be optimized for these steps, which are testing and deployment processes. This results in a higher prioritization of specific non-functional requirements, which must be taken into account when architecting. Practices may be to introduce APIs for testing and/or make use of principle 4 and test loosely coupled components independently. Principle 6 suggests an organizational structure orientated by capabilities/user stories, rather than by layers of the architecture. Organizing the development teams in such a way, enhances the feeling of responsibility for a functionality and ensures independence between development teams.

In this research continuous architecting is defined as a set of architecting rules and supporting tools to efficiently support both incremental and agile delivery methods, like Continuous Delivery, from an architectural perspective [6]. It is neither a process description nor a methodology [6].
This makes this perception independent from any domain as long as an incremental or agile delivery method is applied. One aspect shared by perception I and III is the idea of delivering a minimum viable architecture. Perception II and III both aid agile (or in some cases incremental) development/delivery practices, but differ in their goals: Perception II focusses on long-term responsiveness, whereas this one focusses on improving software delivery. Although not defined as continuous architecting approaches, two researches with similar approaches from 2015 promote the idea behind this perception of continuous architecting. Let's see if it is actually a new approach and what we can additionally learn from similar approaches. The first research is by Lianping Chen [4] and deals with the challenge to integrate existing products and projects to Continuous Delivery process and the new challenges Continuous Delivery has given architecting practices. The main research question is what architecting for Continuous Delivery implies; more details on the concept can be found in the referenced paper [4]. Outcome of this research are a set of architecturally significant requirements (ASR). These are non-functional requirements that must be met by architectures to successfully integrate the product into the Continuous Delivery process. Modifiability, Deployability and Testability are part of the mentioned ASRs, which are related to the above mentioned principles 2, 4 and 5 by Erder and Pureur [6]. These ASRs promote the acceleration of product delivery, since they ensure that the architecture is optimized for the different stages of the development life cycle.
The second research by Michael Waterman *et al.* [8] deals with the issues of how much architecting should be done up-

front of agile software development. They present a guideline to handle this issue as result of a grounded theory of agile architecting. This guideline consist of five architecture strategies. These strategies can be applied depending on the architecture team's context, which consists by six forces (e.g. Experience or Team Culture) [8]. More details on the concept can be found in the related paper [8]. Important is what we can learn from this approach: Early delivery in context of Continuous Delivery can only be achieved by reducing upfront effort and neglecting some future-features [8]. This includes neglecting architectural practices, and therefore means to delay design decisions (related to principle 2 [8]). Design decisions which are not made upfront cause less architecturally technical debt. Instead of a full architecture and analog to a minimum viable product [6], a minimum viable architecture should be developed. ASRs are also mentioned in this paper as a cause for architecture complexity. Therefore these requirements are high prioritized.
So parts of perception III approach have already been proposed under the name of agile architecting. This yields the same discussion need as in the last subsection. But since continuous architecting here is not defined as a process or a methodology, where for example Waterman *et al.* [8] rather give a preset of when to apply which strategy.
Therefore it can be assumed that this perception's method is also new although the similarity to Agile Architecting cannot be overseen. Also, this perception exceeds the two promoting researches, since more principles are introduced.

Concluding it can be said that continuous architecting was defined as a set of rules of an architecting style that aids in speeding up software delivery and is similar to two Agile Architecting approaches. Also, rough similarities are shared with the preceding perceptions.

4.4 Perception IV

The micro-service architecture was shortly mentioned in the last subsection and is a discussed topic in an continuous architecting online blog [9], wherefore this becomes a perception of its own. Micro-services are definitely not a new architecture model, as they were introduced in agile development around the early 2000 and the fundamental idea is even older. The micro-service architecture derives from the unix philosophy: The architecture consists of many small components, which are loosely coupled, run independently and only implement one task, for which they are specialized. The micro-service architecture is a special version of perception III because it is conform with the six principles mentioned. It specializes in principle 4, due to all components being loosely coupled. Also, the architecture is fit for build, test and deployment steps, since components are independent from each other and have their own runtime environment.
Regarding all so far presented approaches, this approach is rather static. Where all others present continuous architecting as something dynamic that produces an architecture description over time, micro-services seem like an out-of-the-box solution. The architecture design is already given and does not have to be reinvented. One interesting question is if the micro-service architecture can solve all problems that are presented in perception III sufficiently, as it is more a predevelopment blueprint, which originally was a core problem.
Micro-services are well suited for an agile development and delivery process. Every service is mostly independent of the rest of the system, which is why new services are easily introduced into a running system. They can be implemented in different languages, as long as the meet certain fixed interface requirements. Parallel work on features is possible. Each service can be tested individually. A micro-service architecture is defined as robust, because failure in one service does not force the whole system to fail.
Therefore, this architecture style is not suitable for domains that are forced to implement heavy coupled components, for whatever reason. Also, due to the fixed architecture design, maybe some non-functional requirements are determined and/or trade-offs between them might be restricted. For example, security is a well known problem with micro-services. The attack surface is enlarged by communications growth, hence this implies a growth of interfaces, exposing software functionality. Security for micro-services can hardly be assured without a trade-off for other non-functional requirements. This hinders the responsiveness concerning change in quality requirements.
So in perspective of perception III the micro-service architecture is a good solution for delivery acceleration, only if the architecture suits the products quality requirements.

But micro-services can also support other continuous architecting perceptions. Eberhard Wolff presents two understandings of continuous architecting in his online blog [9]:

1. The phrase continuous architecting derives from the phrase Continuous Delivery and describes improvement in deployment processes

2. Architecture is no document but rather a process and the architecture must be adjusted continuously according to new experiences gained by developers and by architects and changing requirements.

Both definitions are familiar.
As micro-services where already discussed in context of Wolff's first definition, they shall now be discussed regarding the second, which resembles perception II. To recall, this deals with communication issues in companies hindering long-term value delivery. With the micro-service architecture applied communication need shrinks within the group working on a product [9]. This is due to every service being runtime-independent from the others. Freedom in decision-making in development teams includes details as choosing a framework or a programming language. Different development teams do not need to know the implementation of other services as long as they have knowledge of their interfaces. What is new about this statement is, that the services are completely runtime independent software components and not for example classes in the same executable. This implies a new context for architecting practices [9]. The architect only needs to design the communication protocols and assign the functionalities to be implemented by a service [9]. The architect does not dictate how developers have to implement the said functionality for a service anymore. So communications issues as mentioned in perception II are resolved by this architecture. Long-term responsiveness is also assured, due to a sound architecture design.

The micro-service architecture as a static concept also brings a solution to the other continuous architecting perceptions. Back to the research questions: Eberhard Wolff's definitions of continuous architecting are already known from perceptions II and III , i.e. no new definitions are introduced here. But new methods to resembling definitions of perception II and III. The micro-service architecture, if it suits the products quality requirements, is a good solution to architecting challenges but must be handled with care considering significant changes in quality requirements.

5. DEFINITION OF CONTINUOUS ARCHITECTING

To formulate a final answer to the first research question the found definitions are reviewed. Although there were only four definitions for continuous architecting, some are mostly similar. Due to the small amount of found definitions, there is no definition that forms the majority. Without further detail the definition of continuous architecting remains twofold:

1. Continuous architecting is the continuous improvement of an architecture by including new experiences, gained by developers and architects throughout development, and responding to changes in requirements via applying dynamic architecture development and management practices with the aid to support agile software development practices.
2. Continuous architecting is the elimination of bottlenecks in the software development life cycle by improving delivery processes in incremental or agile delivery methods via adopting an architectural style that optimizes the architecture for test and deployment scenarios.

Continuous architecting remains a generic term for software architecture improving practices, due to these ambiguous definitions.

Now an answer for the second research question shall be given. Both of the above definitions are similar to different approaches that are already defined as Agile Architecting. Therefore it can be said that continuous architecting shares the same aims as Agile Architecting, although Continuos Architecting brings forward some new practices in comparison to the Agile Architecting papers and researches presented here.

6. CONCLUSION

The perception's approaches turn out to be a technical framework aiding big-data architecture design, a conceptual framework that restructures organizations to improve long-term value delivery, a set of rules or guidelines that describe an architecture style, which speeds up the software delivery process, and finally a special architecture model, the micro-service architecture. The need for solving only a small subset of a large set of architectural problems, results in very different kinds of approaches, depending on the problems that are solved. As mentioned in the previous section, there are only vague definitions for a subset of approaches that are labeled as continuous architecting.

Without further effort of giving a concrete definition, continuous architecting and Agile Architecting, as terms, may be used interchangeably, meaning that continuous architecting is just another buzzword.

7. FUTURE WORK

Since this research is only based on so little amount of papers, future work could include evaluating future perceptions or researches on continuous architecting to expand this research. On the other hand the authors of the presented papers could specify their understanding of continuous architecting by giving a more precise definition themselves.

8. REFERENCES

[1] ISO/IEC/IEEE 42010 A Conceptual Model of Architecture Description. `http://www.iso-architecture.org/ieee-1471/cm/`. Accessed: 2016-12-10.

[2] ISO/IEC/IEEE 42010 Thoughts on an Architecting Process. `http://www.iso-architecture.org/ieee-1471/applying-the-standard.html`. Accessed: 2016-12-10.

[3] M. M. Bersani, F. Marconi, D. A. Tamburri, P. Jamshidi, and A. Nodari. Continuous Architecting of Stream-Based Systems. *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 146–151, 2016.

[4] L. Chen. Towards Architecting for Continuous Delivery. *Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015*, pages 131–134, 2015.

[5] T. Dybå and T. Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, 2008.

[6] M. Erder and P. Pureur. *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric-World.* 2016.

[7] A. Martini and J. Bosch. A multiple case study of continuous architecting in large agile companies: current gaps and the CAFFEA framework. *Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016*, pages 1–10, 2016.

[8] M. Waterman, J. Noble, and G. Allan. How much up-front? A grounded theory of agile architecture. *Proceedings - International Conference on Software Engineering*, 1:347–357, 2015.

[9] E. Wolff. Continuous Architecting Blog. `https://www.heise.de/developer/Continuous-Architecture-2687847.html`. Accessed: 2016-12-11.

A Look at the Evolution of Software Architecture Evolution since 2010

Joël Pepper
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
joel.pepper@rwth-aachen.de

Ana Nicolaescu
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
ana.nicolaescu@swc.rwth-aachen.de

ABSTRACT

As Software systems have become increasingly more complex and longer-lived, the focus of Software Development has started to shift towards not only architecting software well, but keeping it well-architected throughout its life span. This continuous adaptation of software architecture over years of changing needs and goals is called software architecture evolution (SAE). Understanding the ramifications of SAE will become only more important as the trends toward even more complex and even longer-lived software systems continues.
In 2012, Breivold et al. published a comprehensive systematic review of the existing research into SAE [7]. The review categorizes and analyses the research available in major electronic databases with publishing dates before end of August 2010. This paper provides a look at how the research field itself evolved since then. To assess the direction SAE research has taken since 2010, we analysed the impact each of the 82 studies identified in Breivold's review has had now that 6 years have passed. Further we compared the studies current impact with their citation ranking that Breivold identified in 2010, showing which studies gained traction in the SAE research community and which were disregarded. Finally, we partially replicated Breivold's review for the time span of 2010–2016. We employed Breivold's methodology to identify 5 of the most influential papers published between 2010 and November 2016. We then used these studies as representatives to gauge which research trends were the most common during this time span. By comparing the impact of these newer papers with the most influential papers Breivold identified in 2010 we established their overall relevance in the field. Based on the results of the impact-analysis, we then formulated multiple plausible hypotheses to explain trends visible in the data. We also discussed possible ways of falsification for each hypothesis as basis for deeper analysis in further research.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

software architecture evolution, software architecture, systematic review, literature review

1. INTRODUCTION

The research into Software architecture has made great strides toward enabling software engineers to create complex software systems in an increasingly more reliable and well understood fashion. However, now that these software systems underpin almost every facet of our lives, the more important systems have gained a lot of inertia. This requires them to stay in place and cope with an ever changing environment as "legacy systems", due to the infeasibility of outright replacement. Modern software systems often have lifespans in the range of 10–30 years [20, 6]. No matter how well the software architecture fulfils all requirements and enables maintainability, it will need to evolve to satisfy new goals, concerns and opportunities, sometimes in significant ways. Understanding these pressures on software architecture, which go beyond the concept of maintainability, forms the basis of Software Architecture Evolution (SAE) research.

Because longer-lived software systems have only become widespread in the last few decades, the field of Software Architecture Evolution is relatively young. Most research has been conducted in the last 15 to 20 years. In an important step into understanding the extent of the field, Breivold et al. [7] have conducted a systematic review of the field, identifying 5 main categories of Software Architecture Evolution research. Breivold's review looked at all studies published before the third quarter of 2010. While this review has been very thorough and helpful in comprehending the existing research on Software Architecture Evolution, almost 6 years have passed since the data was gathered and analysed by Breivold et al. For such a young subject of research 6 years can represent a very long time in which major shifts and breakthroughs can happen.

We believe that updating the review is necessary to once again gain a complete picture of the field. While we are not able to completely re-do Breivold's review using the full set of all matching studies published to date, we will re-examine the original set of relevant studies identified by Breivold et al. as well as identify 5 high-impact studies published between 2010 and 2016. These studies will serve as representatives of the direction SAE has evolved since Breivold's review. By comparing relative impact of this new dataset as well as changes in Breivold's set of 82 studies to Breivold's original impact analysis, we can formulate plausible hypotheses about the reason why these particular studies were favoured by the research community. This in turn could allow us to discern the current influences on and directions of SAE research. Such hypotheses can then serve as research question to evaluate the current State of Software Architecture Evolution in depth in follow up studies. As such we view this paper akin to a pre-study that enables us to formulate targeted, potent research questions for indepth study of the complete set of all studies.

The remainder of this document is structured as follows: Section 2 outlines the methodology we employed and how we developed it, divided into the areas of impact analysis, data extraction from Breivold's results and data collection with regards to finding recent influential studies. In Section 3 we describe the results of our data extraction ordered by citation count, comparing previous citation rankings recorded by Breivold [7, Table 5] with our current ranking. We also present the results of our data collection and put the relative impact of the current papers in perspective with the extracted studies from [7]. In Section 4 we analyse the trends that emerged in the data and put forth multiple plausible hypotheses as to why these trends can be observed. Finally, Section 5 collects our considerations, conclusions as well as starting points for future research.

2. METHODOLOGY

Because our study was created as an extension of Breivold et al.'s systematic review, which in itself is based on the guidelines outlined in by Kitchenham et al.[15], we took specific care in constructing our research protocol to preserve comparability with Breivold's review. We worked with two main datasets: The 82 studies originally identified by Breivold (which we will refer to as Breivold82 in the rest of the paper) and a small set of representative high-impact studies published since 2010. Both sets of studies were collected in the citation tool Mendeley[1] to facilitate collaboration and review of the dataset as well as simplify organization of the studies.

2.1 Impact Analysis

We decided on aggregate number of citations of all published versions as a measure of impact for a given study. The reasoning for this was twofold: First citation count is a generally good, although not perfect indicator of impact as long as a comprehensive list of sources is used to find citations. Second citation count based rankings provide comparability to the results of [7, Table 5], which analysed citation counts as of 2010. The aggregate citation counts were taken from Google Scholar [2] as of November 30th 2016.

2.2 Breivold82

The full dataset Breivold et al. identified as relevant to the field of Software Architecture Evolution is listed as references in the appendix of their paper [7]. The data is represented as a list of human readable references. The references are formatted in a consistent style, which starts with a comma-delimited List of Authors, followed by the Title, the containing publication, the year of publication and a page reference where applicable. The format employed does not conform to any of the popular "MLA", "APA", "Chicago", "Harvard" or "Vancouver" citation styles. Furthermore, the same character (a comma) is used to delimit the fields of the reference as well as the entries of the Author list. This uncommon format made it impossible to parse the reference string automatically with common freely available reference extraction tools. Writing a parser for the custom format employed would have been possible, however we opted for human extraction via manual searches on Google Scholar, as this allowed us to also retrieve the desired aggregate citation counts for all versions of each paper at the same time. Having extracted the data set and connected every item to a Google-Scholar backed citation count allowed ordering of the set and comparison to Breivold's most cited studies ranking from 2010 [7, Table 5]).

2.3 Recent High-Impact Studies

In order to replicate Breivold's review for papers published since 2010, we queried the seven electronic study databases listed in [7, Section 2.2] using the search terms outlined in [7, Section 2.3] for initial discovery[3]. However, we were not able to search the "Compendex"[4] database, due to insufficient access privileges. To compensate for this, we included a search on Google Scholar with the same query, as Google Scholar is able to index the missing database and should therefore be able to discover all relevant studies contained in the "Compendex". The results were filtered to exclude studies published before 2010 using each database's internal filter mechanism. Because we were only interested in identifying a few representative studies with high impact, we did explicitly not test every single item for its adherence to the inclusion and exclusion criteria outlined in [7, Table 1]. This approach would have been infeasible, given the scope of this pre-study, as most of criteria cannot be evaluated automatically. Instead, we used the database's internal citation counts to order the results in descending order. From every database, we obtained the two most cited studies, which fulfilled the inclusion and exclusion criteria based on Title and Abstract of the paper. This process mirrors steps (i) to (iii) of the multi-step process used by Breivold et al. for study selection. These studies formed the initial discovery set, from which we selected the 5 most influential studies according to our criteria for (see Impact Analysis) for the final comparison against the most cited studies among the Breivold82 (as identified by Breivold in 2010 [7, Table 5]).

3. RESULTS

We first collected and sorted our results for each data set separately. Then we merged the 5 most influential recent high-impact studies and surveyed the combined data set after ordering anew.

[1]`https://www.mendeley.com`

[2]`http://scholar.google.com`

[3]SpringerLink articles which were marked "preview-only" and therefore unpublished as of November, 30th 2016 were not included in the dataset, because they are not technically published yet at the time of writing

[4]`http://www.engineeringvillage.com`

Rank	Citations	Change	Title
01	7081	0	L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley Professional,2010
02	2493	0	L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos, Non- functional Requirements in Software Engineering, Springer, 2000.
03	1943	0	J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach, Addison-Wesley Professional, 2000.
04	1266	0	P. Clements, R. Kazman, M. Klein, Evaluating Software Architectures: Methods and Case Studies, Addison- Wesley, 2006.
05	989	0	C. Hofmeister, R. Nord, D. Soni, Applied Software Architecture: A Practical Guide for Software Designers, Addison-Wesley Professional, 2000.
06	664	+2	M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, W.M. Turski, Metrics and laws of software evolution – the nineties view, in: 4th International Symposium on Software Metrics, 1997
07	640	-1	R. Kazman, L. Bass, G. Abowd, M. Webb, SAAM: a method for analyzing the properties of software architectures, in: International Conference on Software Engineering, 1994, pp. 81–90.
08	612	-1	R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, The architecture tradeoff analysis method, in: IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 1998, pp. 68–78.
09	334	+2	P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet, Architecture-level modifiability analysis (ALMA), Journal of Systems and Software 69 (2004) 129–147.
10	330	0	K.J. Sullivan, W.G. Griswold, Y. Cai, B. Hallen, The structure and value of modularity in software design, in: European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2001, pp. 99–108.
11	329	-2	M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, H. Lipson, Attribute-based architecture styles, in: Working IEEE/IFIP Conference on Software Architecture (WICSA), 1999.
12	261	*	E. Fricke, A.P. Schulz, Design for changeability (DfC): principles to enable changes in systems throughout their entire lifecycle, Journal of Systems Engineering 8 (2005).
13	204	-1	R. Kazman, J. Asundi, M. Klein, Quantifying the costs and benefits of architectural decisions, in: Interna- tional Conference on Software Engineering, 2001.
14	160	*	A. Tang, M. Ali Babar, I. Gorton, J. Han, A survey of architecture design rationale, Journal of Systems and Software 79 (2006) 1792–1804.
15	143	*	A. Tang, P. Avgeriou, A. Jansen, R. Capilla, M. Ali-Babar, A comparative study of architecture knowledge management tools, Journal of Systems and Software 83 (2009) 352–370.
16	139	*	M. Ali Babar, I. Gorton, A tool for managing software architecture knowledge, in: International Conference on Software Engineering Workshop on Sharing and Reusing architectural Knowledge-Architecture, Rationale, and Design Intent, 2007.
17	137	-4	P. Bengtsson, J. Bosch, Architecture level prediction of software maintenance, in: European Conference on Software Maintenance and Reengineering (CSMR), 1999, pp. 139–147.
18	134	-1	A. Jansen, J. Van der Ven, P. Avgeriou, D.K. Hammer, Tool support for architectural decisions, in: Working IEEE/IFIP Conference on Software Architecture (WICSA), 2007.
19	130	-4	W.M.N. Wan-Kadir, P. Loucopoulos, Relating evolving business rules to software design, Journal of Systems Architecture 50 (2004) 367–382.
...			
26	84	-10	N. Lassing, P. Bengtsson, H. van Vliet, J. Bosch, Experiences with ALMA: architecture-level modifiability analysis, Journal of Systems and Software 61 (2002) 47–57.
...			
58	23	-44	P. Bengtsson, J. Bosch, Scenario-based software architecture reengineering, in: International Conference on Software Reuse, 1998, pp. 308–317.

Table 1: Excerpt of the 82 Software Architecture Evolution studies identified by Breivold's review [7] ordered by number of citations obtained from Google Scholar as of November 30th, 2016. The second column identifies the shift in position compared to Breivold's original ranking [7, Table 5] according to number of citations obtained from Google Scholar as of 4th of September, 2010. An asterisk (*) indicates that the study had no former rank in Breivold's original Top-17 ranking. Studies which have increased in relative impact are coloured green, while those which decreased are coloured red. Studies without previous rank, i.e. studies which were below rank 17 in 2010, are highlighted in beige. Unchanged entries are coloured in grey.

Database	Matches
ACM Digital Library	2355
Compendex	-
IEEE Xplore	5243
ScienceDirect - Elsevier	2020
SpringerLink	1440
Wiley InterScience	629
ISI Web of Science	247
Google Scholar	N/A

Table 2: The number of matching studies found in each electronic database using the binary search string outlined in Recent High-Impact Studies as of November 30th, 2016. No results for Compendex could be retrieved, due to insufficient access privileges. SpringerLink articles which were marked "preview-only" and therefore unpublished as of November 30th 2016 were not included in the dataset, because they are not technically published yet at the time of writing. Google Scholar's estimated number of result varies strongly between repeated queries and is known to be unreliable and was therefore omitted.

3.1 Breivold82

Table 1 shows the Breivold82 ordered by number of citations obtained from Google Scholar as of November 30th, 2016, including changes in ranking for the 17 studies which were listed in [7, Table 5], by Breivold et al. as the 17 most-cited studies amongst their dataset. Unfortunately, Breivold et al. did not provide a ranking past the Top-17, as such we do not know what relative impact these other studies had at the time the data was gathered. It should be noted, that the Top 5 studies have not changed in ranking since 2010 and that the 11 most cited studies have remained the same, except for reordering among the ranks 6–11. The first "new" study, that is a study that was not among the 17 most-cited Software Architecture Evolution studies in 2010, and therefore had not been captured in Breivold's table of most-cited studies, now appears on Rank 12 within the Breivold82. Position 13 is held by former Rank 12 and Rank 14–16 are filled with three studies which also did not appear in Breivold's 2010 ranking. Rank 17–19 are taken by studies which have been displaced by the newcomers. There are only two studies which are not close to their former rank. The first is [18], which has dropped 10 places and now occupies rank 26. Biggest "Loser" is [3] which has been overtaken by 44 other studies and now resides on rank 58.

3.2 Recent High-Impact Studies

Table 2 shows the number of matching papers identified in each queried database using Breivold's search String. In the pre-selection phase, 14 Papers were identified, two from each individual database, according to the database's internal citation counts for each study. As noted in the Methodology section, we replaced the query to the Compendex with a Google Scholar query, because Google Scholar can index the Compendium and therefore this query will yield two papers that are at least as influential as the two most influential papers available in the Compendex. Table 3 shows the 14 papers broken down by publishing year.

3.3 Comparison to Breivold's Results

Table 4 lists the five most influential studies taken from among the initial discovery set of 14 listed in Table 3 which were found by applying Breivold's [7] search criteria to the time span of 2010–2016. After adding these five to the initial Breivold82 set, we resorted the studies by citation count to generate a comparative ranking. The first column of Table 4 lists the position of each study in this expanded ranking, as recorded by Google Scholar as of November 30th 2016.

4. DISCUSSION

4.1 Breivold82

When we look at the changes within the Breivold82 set from 2010 to 2016, we immediately noticed the block of 5 studies near the top, which have kept their place as most cited SAE studies of all time and whose citation counts set them far apart from the rest of the dataset. The single most cited study ([2]) alone has been cited more often than any study outside of the Top 5 combined (7081 citations vs. 6624 combined citations). A striking commonality between these Top 5 is that all of them are standalone published books rather than singular studies published in technical journals. Because such books are far more comprehensive than reports and papers which tend to be focused on much more narrow scopes, and often include definitions for basic terms and concepts it is expected that these books would aggregate a large number of citations. It is, however, undeniable that these have had a strong impact on SAE research and understanding the concepts, which are discussed in the Top 5 Books is likely essential to understanding the current state of SAE research. Next in the ranking is a cluster of three studies with similar citation counts of 664, 640 and 612. The comparison to Breivold's data indicates that the paper by Lehman [19] has overtaken the two papers by Kazman [13] [14] but only by a small margin. Because this three-study cluster has kept its position in terms of relative impact and there does not seem to have been a major upheaval within it, we think that Breivold's initial analysis of the three studies should still apply as well as it did in 2010. Next on the ranks 9–11, we find another cluster of three studies by Bengtsson[5], Sullivan[21], and Klein[16], with even more closely grouped citation counts: 329, 330 and 334. With such little difference in citation count, relative ordering within the cluster can be easily dismissed as irrelevant. The cluster itself has remained in the same position compared to Breivold's original results, therefore we again defer to Breivold's original analysis of the studies as it still applies. On rank 12 lies the first of the "new" studies, that is a study whose relative impact had placed it at rank 18 or lower in 2010 and was therefore not captured by Breivold's most cited studies [7, Table 5]. The paper by Fricke et al.[9] outlines principles focused on bolstering the aspects of flexibility, agility, robustness and adaptability in systems architecture design to enable changeability. The gain in impact for this study since 2010 points to an increase in importance and interest for what Breivold identifies as the category of "Influencing factor focused quality considerations during software archi-

2010	B. Williams, J. Carver, Characterizing software architecture changes: A systematic review
	E. Jackson, E. Kang, Components, platforms and possibilities
	H. Tajalli, J. Garcia, PLASMA: a plan-based layered architecture for software model-driven adaptation
	R. Lagerström, P. Johnson, Architecture analysis of enterprise systems modifiability — Models, analysis, and validation
	Q. Zhu, Y. Yang, Optimizing the Software Architecture for Extensibility in Hard Real-Time Distributed Systems
	S. Bode, M. Riebisch, Impact Evaluation for Quality-Oriented Architectural Decisions regarding Evolvability
	T. Bellizio, G. De Tommasi, The Software Architecture of the New Vertical-Stabilization System for the JET Tokamak
	M. Greiler, H. Gross, Understanding Plug-in Test Suites from an Extensibility Perspective
2011	D. Budgen, A. Burn, Empirical evidence about the UML: a systematic literature review
	G. Reggio, E. Astesiano, A Problem Frame-Based Approach to Evolvability: The Case of the Multi-translation
2012	H. Mannaert, J. Verelst, Towards evolvable software architectures based on systems theoretic stability
	J. Conejero, E. Figueiredo On the relationship of concern metrics and requirements maintainability
	M. Mirakhorli, Y. Shin, A tactic-centric approach for automating traceability of quality concerns
2014	On the Need for Evolvability Assessment in Value Management

Table 3: Breakdown of publishing years of the 14 studies identified in the initial discovery set for influential SAE studies published in 2010–2016. The 5 most influential studies from Table 4 are colored in beige

Rank	Citations	Title
24	91	B. Williams, J. Carver, Characterizing software architecture changes: A systematic review
28	78	D. Budgen, A. Burn, Empirical evidence about the UML: a systematic literature review
39	53	E. Jackson, E. Kang, Components, platforms and possibilities
40	52	H. Tajalli, J. Garcia, PLASMA: a plan-based layered architecture for software model-driven adaptation
43	46	R. Lagerström, P. Johnson, Architecture analysis of enterprise systems modifiability - Models, analysis, and validation

Table 4: The 5 most influential studies from the initial Discovery Set of 14 the found by applying Breivold's [7] search criteria to the time span of 2010–2016. The first column lists the position of each study if the 5 were to be added to the Breivold82 set.

tecture design". Rank 13 ([12]), rank 17 ([4]), rank 18 ([11]) and rank 19 ([25]) are taken by studies from the 2010 Top 17 which have been slightly displaced by the arrival of newcomers to the top of the ranking, however the loss in relative impact is rather small and these studies still hold similar importance to that of 2010, as such we again defer to Breivold's initial analysis. More interesting, however, is the cluster of three studies [24],[23] and [1] on ranks 14–16, all of which are new to the Top 17 and share a common Author: M. Ali Babar and which focus on the topic of architecture knowledge management, which Breivold previously identified as one of the 5 main categories of Software Architecture Evolution. This is a very strong indicator that the category of architecture knowledge management has gained importance in the SAE research community in recent years.

4.2 Recent High-Impact Studies

The publishing year breakdown of the 14 studies taken from the electronic databases for the time period of 2010–2016 presented in Table 3, shows the discovery set skewing towards older studies. This trend is further exemplified if we look at the age of the five most influential studies in the set. Of the five, four were published in 2010, the earliest year captured by the search and only one study was published not in 2010 but 2011. No studies published after 2014 had garnered enough impact as of November 2016 to be even captured in the discovery set. This strong correlation between citation count and age calls into question the validity of discovering recently published influential papers by citation count alone. However the fact that the discovered studies have gained enough citations to be ranked at the top of the second quartile if ranked together with the much older studies in the Breivold82 set, clearly indicates that these Top 5 are influential studies, which are likely to rise even further in influence, given similar time frames to the studies in Breivold82. As such we are convinced, that they can serve as representatives of newly developed and developing trends in the field of Software Architecture Evolution.

The first and most influential, of the current studies is a systematic review conducted by Williams et al. [26], which aims to characterize and categorize the changes that can be made to software architecture. A robust categorization assists in assessing the impact any given change to a software architecture has. Understanding these impacts, especially in the long term, is very helpful in understanding and steering the evolution of a software architecture and is therefore of obvious interest to SAE research. The emergence and popularity of such a categorization in recent years suggests that the tools and methods to quantify changes in software architecture were lacking, which would have had a negative effect on the progress of SAE research. A review, focused specifically on efforts to categorize and quantify changes in software architecture, could validate or falsify this postulation.

The second paper, another literature review, by Budgen et al. [8] focusses on the topic empirical research about the effectiveness of the Unified Modelling Language (UML). UML has become the de-facto standard object-oriented modelling language in many areas of software development. The popularity of this study serves as an indicator for the importance of UML to the topic of SAE and software architecture in general, giving software architects and developers a commonly accepted tool to model software. This claim could be further substantiated by analysing the role that UML has played in major case studies of SAE.

Both the third [10] and fourth [22] most influential recent studies are concerned with the topic of Model-Driven Architecture (MDA), specifically in how these architectures adapt to changing requirements. [10] describes a framework for the automation of MDA, where deliberate changes to the architecture by designers and architects is concerned, while [22] outlines a plan-based layered architecture that harnesses MDA to enable self-adaptation in software systems. The prominence of these two papers suggests, that MDA has been an active topic of SAE research in recent years, which could be verified by an analysis focused on papers about the topic of MDA.

The last of the representative papers, by Lagerström et al.[17], deals with modelling complete Enterprise Architectures (EA) as opposed to only focussing on architectures of specific software systems. The paper analyses the use of Probabilistic Relational Models (PRMs) in a formalized EA analysis approach. A big focus is given to the modifiability aspect of the model. As such, we can group it with the previous two papers which also deal with modelling architecture change although on a slightly smaller scale. This leads us to believe, that the category that Breivold et al.[7] describe as "Modelling techniques" is of special import in the current climate of SAE research. A way of verifying this hypothesis would be a review of the literature with keywords like "Model-driven architecture" or "enterprise architecture analysis" as well as tracing the impact of the papers Breivold et al. sorted into the category of "modelling techniques"

5. CONSIDERATIONS, CONCLUSIONS AND FUTURE RESEARCH

We were only able to survey the body of work in the field of Software Architecture Evolution since 2010 in a somewhat superficial manner, choosing representative papers to gauge trends and interests in the whole field. Although citations recorded by Google Scholar were used to record the final citation data for each paper, the pre-selection of representative papers used the databases' internal citation counts to find influential papers. This was necessary, because we did not have the resources to process the complete dataset of 10638 results manually. Making use of the sorting functionality built into the interface of each individual database, provided a more objective and reproducible method of pre-selection. For future, more in-depth examination of the post-2010 period, we would like to repeat the data collection phase by writing a parser application. Such a parser would be able to join the result sets from all databases and connect each item via means of DOI to the correct Google Scholar citation data. The assembled data set could then be sorted and filtered without the possibility of being over-shadowed or disregarded due to the less complete citation information available to the separate database sorting functions. This method would also negate the possibility of the dataset being influenced by potential quirks, bugs and oddities in the sorting functions employed by each individual database. Using only a single consistent data source would make any of these problems systemic and easier to spot and therefore to negate or mitigate their effect. Furthermore, while access to these databases was kindly provided by the RWTH Aachen Universitätsbibliothek, due to time and resource constraints we were not able to obtain and evaluate the full text of each paper. Negotiating and/or buying access to the full text of all representative papers could and should occur as part of the falsification process for the hypotheses we proposed in the Discussion section in follow up studies.

6. REFERENCES

[1] M. A. Babar and I. Gorton. A Tool for Managing Software Architecture Knowledge. In *Second Workshop on Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent (SHARK/ADI'07: ICSE Workshops 2007)*, pages 11–11. IEEE, may 2007.

[2] L. Bass, P. Clements, R. Kazman, and A. Wesley. Software Architecture in Practice, Second Edition. 2003.

[3] P. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 308–317. IEEE Comput. Soc, 1998.

[4] P. Bengtsson and J. Bosch. Architecture level prediction of software maintenance. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*, pages 139–147. IEEE Comput. Soc, 1999.

[5] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, jan 2004.

[6] H. P. Breivold, I. Crnkovic, and P. J. Eriksson. Analyzing software evolvability. *Proceedings - International Computer Software and Applications Conference*, pages 327–330, 2008.

[7] H. P. Breivold, I. Crnkovic, and M. Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40, 2012.

[8] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical evidence about the UML: a systematic literature review. *Software: Practice and Experience*, 41(4):363–392, apr 2011.

[9] E. Fricke and A. P. Schulz. Design for changeability (DfC): Principles to enable changes in systems throughout their entire lifecycle. *Systems Engineering*, 8(4), 2005.

[10] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, platforms and possibilities. In *Proceedings of the tenth ACM international conference on Embedded software - EMSOFT '10*, page 39, New York, New York, USA, 2010. ACM Press.

[11] A. Jansen, J. Van Der Ven, P. Avgeriou, and D. K. Hammer. Tool support for Architectural Decisions. 2007.

[12] R. Kazman, J. Asundi, and M. Klein. Quantifying the costs and benefits of architectural decisions. In *Proceedings of the 23rd international conference on Software engineering*, pages 297–306. IEEE Computer Society, 2001.

[13] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. 1994.

[14] R. Kazman and M. Klein. The architecture tradeoff analysis method. *Fourth IEEE International Conference on Engineering of Complex Computer Systems*, (July):68 – 78, 1998.

[15] B. Kitchenham. Procedures for Performing Systematic Reviews. 2004.

[16] M. Klein and R. Kazman. Attribute-Based Architectural Styles. 1999.

[17] R. Lagerström, P. Johnson, and D. Höök. Architecture analysis of enterprise systems modifiability âĂŞ Models, analysis, and validation. *Journal of Systems and Software*, 83(8):1387–1403, 2010.

[18] N. Lassing, P. Bengtsson, H. van Vliet, and J. Bosch. Experiences with ALMA: Architecture-Level Modifiability Analysis. *Journal of Systems and Software*, 61(1):47–57, mar 2002.

[19] 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. 1997.

[20] C. D. Rosso and A. Maccari. Assessing the architectonics of large, software-intensive systems using a knowledge-based approach. 2007.

[21] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The Structure and Value of Modularity in Software Design. 2001.

[22] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic. PLASMA. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 467, New York, New York, USA, 2010. ACM Press.

[23] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar. A comparative study of architecture knowledge management tools. *Journal of Systems and Software*, 83(3):352–370, mar 2009.

[24] A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of architecture design rationale. *Journal of Systems and Software*, 79(12):1792–1804, dec 2006.

[25] W. Wan-Kadir and P. Loucopoulos. Relating evolving business rules to software design. *Journal of Systems Architecture*, 50(7):367–382, jul 2004.

[26] B. J. Williams and J. C. Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51, 2010.

From EA models to UML

A guideline on how to generate UML diagrams from ArchiMate

Maximilian Peiffer
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
maximilian.peiffer@rwth-aachen.de

Simon Hacks
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
simon.hacks@swc.rwth-aachen.de

ABSTRACT

Because enterprise architects prefer to model their organization as a whole using EA (Enterprise Architecture) models and not UML (Unified Modeling Language), software developers do not get diagrams in UML format which are widely used in the field of software engineering and to which they are used to. Formerly they had to either think about the software solutions already modeled in an EA model and create UML diagrams more or less from scratch or they had to replicate it completely, which is time consuming.
This paper describes an approach in which UML class and sequence diagrams are derived from existing EA models. Because the diagrams are constructed based on already existing and elaborated models, this way is more time-saving and fail-safe than creating entirely new diagrams. With the transformation guideline developed in this paper the software engineers can create UML diagrams from existing structures in EA. Therefore, this paper shows the transformation of sample modules of the application layer of ArchiMate, an open EA-language, to UML class and sequence diagrams. There may be a lack of information, because UML is able to model deeper information than ArchiMate offers. Hence, sources are named from which these additional information can be extracted.

Keywords

Enterprise Architecture, Unified Modeling Language, ArchiMate, Model Transformation

1. INTRODUCTION

Modeling an organization has become very important. To run a successful business, it is eminently important to have a certain strategy or plan[2, p. 179]. To improve this strategy and present it to others a good model is needed. Because of the severity of such a model, many organizations want to model their structure best possible and in one single place. Moreover, they bind everything together and model the organization as a whole. One way to do so is using EA (Enterprise Architecture) models. Every relevant process, resource and relation can be modeled within one model which outlines the "organization of an organization"[1] as well as the design concepts which are dominant[8]. The model can be used as an overview but one can also go into detail if needed. It is possible to pass it to different departments and roles without the need to modify the model. Because the business is presented as well as the technologies and applications used in the organization, stakeholders can be identified. This can be done for example by looking at who is affected by a certain change, who then implement it or who will have to make a decision about it[9]. Thus, in the best case, each stakeholder can extract his specific information.
A problem of this modeling is that enterprise architects usually do not use the same techniques of visualizing as software engineers. Software engineers normally use UML (Unified Modeling Language) diagrams to model their software and pass them to programmers who will implement the software. This is done because these diagrams contain all relevant information needed for the implementation, are easy to read and can be used as a standard. Moreover, an automated code generation from well elaborated diagrams is possible and can save a lot of time. Thus, a transformation from an EA model to UML is needed in order to work with familiar types of diagrams.
In general, there are two main reasons to create a guideline as described in this paper: First, a predefined guideline makes it more easy and fail-safe to transform EA models into UML. Second, there is a lack of research (as described later in section 2) if and how EA models can be transformed to UML diagrams. Only one source[12] can be found that addresses this topic, but it does not go into detail and does not take everything into account EA offers.
One way to model EA diagrams is using ArchiMate, an open EA language. This paper makes use of ArchiMate in version 2.1. ArchiMate structures the organization into the three layers application, technology and business. The business layer contains all business processes performed by business actors which offer products and services to external customers. The application layer defines how these business processes are supported by software applications and the technology layer describes the infrastructure which is used to do so[10]. This leads to the advantage of an easy overview over complex structures.
In this paper the transformation of elements of the application layer is discussed and some shortcomings in a one-to-one transformation are mentioned. Because EA has its focus not on details of software but on the functionality and

embedding of it, some elements needed for the construction of complete UML diagrams are missing and have to be provided from different sources. Three of these sources and how they can be used to obtain the missing information are named.
The structure of this paper is as follows. In section 2 related work is described. Section 3 is about the transformation of selected elements of the application layer to UML diagrams and how the transformation was derived. First, in section 3.1 the composition of a class diagram is described. Second, the transformation to a sequence diagram is depicted in section 3.2. In section 4 the process of generating a class diagram and a sequence diagram from an ArchiMate model is shown using a concrete example. Section 5 is about shortcomings of the described transformations and additional sources that can be used. After section 6, in which conclusions are drawn, the future work on this topic is discussed in section 7.

2. RELATED WORK

At the moment, there are no sources discussing the topic of transforming complete EA models to UML diagrams in detail. Except of Wiering[12], who roughly outlines a mapping from ArchiMate to several types of diagrams in UML 2.0 and The Open Group[11] which gives a vague overview on how a collaboration between ArchiMate and UML can be realized, there is no research in this field. Wiering describes a mapping for each single element of the ArchiMate language and how it can be generally transformed to UML diagrams. He inserts CRUD-operations (create, read, update delete) to each Data Object and Application Component, regardless of if they are actually used. In distinction from this paper, he adds stereotypes like <<*Role*>> or <<*Collaboration*>> to the class diagram in order to represent the original function in the ArchiMate model.
There are many application fields for the topic of model transformation. One well explored field is generating source code from UML diagrams, for example transforming XMI-based UML diagrams to source code using finite state machines as described by Rincon-Nigro et al.[4]. If the rules for the transformations are realized in the finite state machines, they create a reproducible and structured output for a certain input. This approach is very likely to the approach used in this paper, because certain elements from one model are picked and unambiguously mapped to an element of another model. Some of the problems described for the code generation address a lack of information in UML diagrams. For example, sometimes the expected code differs from the generated because there is no way in UML to express certain code fragments[6]. This problem of missing information at the input side also occur while generating UML diagrams from ArchiMate (as described in the next chapter). Because of the similarity of the problem in both transformations, the solutions from the well explored code generation can be adapted.

3. TRANSFORMATION

In this chapter, the transformation of elements of the application layer is discussed. This layer is chosen, because it addresses software applications and thus is best suitable to generate UML diagrams from it. The application layer of ArchiMate contains multiple elements and their relations to each other. Only the most important available concepts are treated here: the constructs application collaboration, application interaction and application service are not regarded in this paper, because they concentrate of the interaction of many components. In this paper, the transformation process is described only for one application component without relations to others.
It is important to mention that there is not only one correct solution to model a certain application in UML[5]. There are several details (e.g. interfaces, access relations and data types as described in the following) that can be modeled one way or the other. To develop a universally valid guideline for the transformation of certain components, the diagrams that result from these transformations are generalized. There may exist a version of these diagrams that reflect a particular situation better but in most cases it is nearly impossible to generate a perfect solution from the ArchiMate model with a single (and simple) guideline. This is due to the fact that ArchiMate was not designed to be transformed to UML and thus does not contain all relevant data. In class diagrams, for example, properties, access privileges and data types are named. These (normally) cannot be found in an ArchiMate model. Sequence diagrams model the temporal and causal sequence of method calls. These sequences are not part of ArchiMate either.

3.1 Class Diagram

In this subsection, the transformation process to generate a UML-class diagram is described. Usually, there are information inside the ArchiMate model that can be used in a class diagram. Hence, the components of such a model and their part in the class diagram is described in the following and can be seen in table 1.
Application component
The definition of an application component in ArchiMate is that it is modular, replaceable and encapsulates its behavior and data[10]. This matches the definition of a class in a class diagram, where the encapsulation of attributes and methods is one of the main criteria for an independent class[5]. Thus, an application component can be mapped to a class in a class diagram.
Application interface
There are two possibilities how the application interface can be modeled in a class diagram. First, as a "normal" interface as used in object oriented languages and second as an implementation of the observer pattern. Which way is most qualified and fits the given purpose best cannot be determined automatically and therefore has to be chosen by the person who initiates the transformation process.
The modeling as an interface is straight forward since interfaces, the "point of access where an application service is made available to a user or another application"[10] are also present in UML[5]. Thus, the application interface can be added to the diagram as an interface and the corresponding relation to the application component has to be drawn. There is nothing that has to be added inside the class or the interface.
The second approach, the observer pattern, models some interaction between application component and interface and also some functionality within the interface. Therefore, two methods and one property is added in order to represent the functionality: The property *interfaces*, holding all interfaces that can be addressed, in the class of the application component which uses the interface, the method *notify()* in the application component class which can notify the inter-

Table 1: Transformation guideline

ArchiMate	UML class diagram	UML sequence diagram
Application Component (AC)	Class	Object
Application Interface (AI)	Interface *or* Observer Pattern (OP): Add property *interfaces*, *register(), unregister(), notify()* to AC, *update()* to AI	Object Add user as new object, add activation call from user to AC to AI If OP: Add *notify()-, update()*-calls at the end
Data Object (DA)	Class	Object
Application Function (AF)	Add function to AC class Add property to DA if one is used	If DA used, add calls from AC and returns from DA: AF with reading access: add *getProperty()*-call AF with writing access: add *setProperty()*-call
Composition, aggregation, association, specialization relation	Same in UML	*(extraneous)*
Used-by relation	Directed association If towards DA: add setter to DA If from DA: add getter to DA	*(extraneous)*
Triggering, flow relation	*(extraneous)*	Arrange calls in triggering/flow order

face to perform its work after certain changes happened and the method *update()* in the interface which stands for the execution of functionality inside the interface. These may be replaced, if there are additional information about the interaction and functioning of application component and interface available. Methods for registration and unregistration of an interface could be modeled as *register()* and *unregister()* methods in the application component class.

Data object
A data object can be used to store certain data. Thus, regarding to the characteristics of a class in UML[5], it can also be mapped to a class. At the beginning, there are no properties and methods available, because the data object in ArchiMate on its own does not tell anything about its functionality.

Application function
An application function describes the internal behavior of an application component[10]. Therefore, the function has to be assigned to the application component as a method or function. Because ArchiMate does not offer data types, return and parameter types are left blank. There is also no information about access privileges so these are also not mentioned in the class diagram.
If an application function makes use of a data object, it is likely that it stores or accesses some data in it. Thus, to express some basic functionality in the class diagram, a property is added to the data object. The name of the property should be the name of the application function to avoid confusion. If the function name consists of a verb and a noun, only the noun is used as property name. If two or more application functions have the same name, numbers have to be added in order to distinguish the sources.

Relations
There are several possible ways how elements in ArchiMate can be connected and related. The composition, aggregation, association, and specialization are taken from UML 2.0[10] and can therefore be mapped one to one.
The association relationship, which is used for a relation that cannot be described by a more specialized relation, is undirected in ArchiMate. In UML, associations are normally directed[5]. Because there is no way to identify the direction, an undirected association has to be used in the resulting class diagram as well. This is also the case for the assignment relation.
The used-by and access relation can be mapped as a directed association to the class diagram. In ArchiMate, the direction of the access relation says if an element has reading or writing access. If the access relation goes from the application function to the data object, the application function has writing access. Thus, to the data object a setter for the property is added. If the relation goes the other way, the application function has reading access and a getter is added. If there is a bilateral access relation between application function and data object, the application function has read and write access and therefore both getter and setter are added. Moreover, if an application component has reading access to a data object, getters for each property are added. Analogous, if an application component has writing access to a data object, setters are added for each property.

3.2 Sequence Diagram

Because ArchiMate does not model many information about the causal or temporal context, deriving a sequence diagram from ArchiMate often has to rely on common concepts instead of modeled relations in EA. The only relation that express temporal or causal dependency is the trigger relation. Thus, if no trigger relation (or flow relation, as described in the following) is available, a sequence diagram cannot be derived just from the information given through the ArchiMate model. This leads to four possibilities: First, all functions of the application component could be displayed one after another as if there was a temporal (one after another) relation between them. This would display the functionality, but is not correct in manner of semantics of the resulting sequence diagram. Second, additional sources (as described in section 5) have to be consulted. Third, the creation of a sequence diagram cannot be accomplished because of the lack of information. The following guideline for transforming ArchiMate elements to an UML sequence diagram is based the fourth possibility: using existing temporal and/or causal structures in the ArchiMate model. Because a sequence diagram models only a certain snapshot of an application there are in most cases many different sequence diagrams possible. This guideline tries to transform the ArchiMate model into one general diagram which displays most of the modeled functionality. An overview can be found in table 1.

Application component

The application component gets its own object and lifeline, because it matches the criteria of a class as mentioned in the previous chapter and can therefore be modeled as such. All functions which are somehow ordered by time in ArchiMate are executed on its activity box in the given order.

Application interface

The application interface is represented as a new object since interfaces are also available in sequence diagrams. As described in the previous chapter, the functions *notify()* and *update* can be added. After the application component called the interface via *notify()*, it calls its *update()*-method in order to perform its functionality. If there is no temporal context available in the ArchiMate model, this is done at the very end of the activity box of the application component in order to keep things simple. To put it at the end seems reasonable because at the end all altered information which can potentially be used are available. If the way of treating the interface as ”normal” interface is chosen, there are no possible actions which can be mapped in the sequence diagram due to a lack of information in the ArchiMate model. If the application component makes use of an interface (regardless which way of representing it was chosen), it is most likely that functions are triggered via the offered interface. Thus, the interface is activated at the beginning of the diagram and creates an activity box in the application component. Because there has to be some interaction with the interface, a user is added who triggers the interface in the first place. Because the exact interaction between user, interface and application component is not known, the calls and returns do not get any naming.

Data object

Because the data object can be treated as class, it also gets its own object and lifeline. Each element which has reading access to the data object can initiate a *getProperty()*-call (where *Property* is replaced by the corresponding property name). Elements with writing access can call *setProperty()* and elements with bidirectional access can do both.
Often the reading of a property cannot be modeled easily because the ArchiMate model does not illustrate when a property is read. However, setting a property can be mapped for example when a function is called which has writing access to the data object. It can be assumed that it can change the value of its property whenever the corresponding function is called.

Application function

The functionality of an application function is executed within the activity box of the corresponding application component. The label of the call is the name of the function. If the function makes use of a data object, the stored information is accessed within the activity box of the function and labeled with either get or set followed by the name of the property corresponding to the function.

Relations

There are two relationships used in ArchiMate which are important for the sequence diagram: triggering and flow.
The triggering relationship implies a temporal relationship where one action is executed after another which can be used in a sequence diagram. If, for example, two application functions are connected by a trigger relationship, they are executed on the same activation box after another.
The flow relationship does not imply a temporal relationship[10]. However, to make the transformation easier, it is treated as a triggering relationship, because if there is a flow expressed, it is likely to say that there also is a temporal order in it.
The access relationship is implemented as a call of the corresponding object. If, for example, an application function makes use of a data object, the call of the data object and its answer is executed in the activation box of the application function (as described above). If an application component uses a data object, the assumptions seems obvious that there has to be a application function that uses the data object as well. Thus, the relation between application component and data object is modeled only via the application function.

4. EXAMPLE

This example is about an electronic sewing machine. It has a display on which the current settings are shown. It is possible to change the form of the stitch (e.g. zigzag stitch to straight stitch). If done so, the user has to enter the stitch width and stitch length for the new stitch. These settings are stored in a data object.
First, the modeling in ArchiMate is described as shown in figure 1. Obviously, the sewing machine is the application component and consists of an application interface (the display). The functionality of the sewing machine is modeled in a parent function called *change Stitchproperties.* Inside this function the stitch form can be changed, which triggers a change of the stitch length and stitch width. All these functions can have writing access a data object called *Stitch* which holds the properties about the currently chosen stitch. The data stored in the data object can be read by the *SewingMachine.* This example is designed in a way so that a class diagram and a sequence diagram can be derived from it. For example, the triggering relationships imply a temporal order which can be used in the sequence diagram.

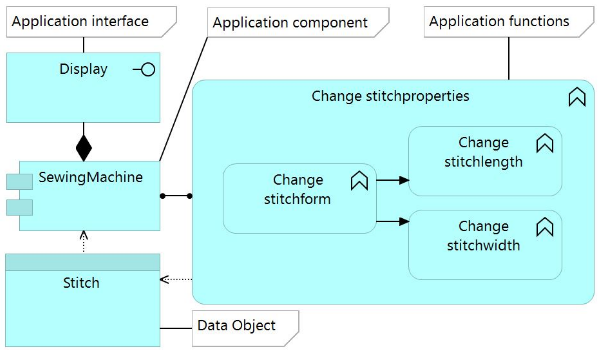


Figure 1: Sewing machine example as ArchiMate model

This model is now transformed into a UML class diagram as described in section 3.1. The first class to be created is *SewingMachine.* The interface can be taken as in ArchiMate. Because in this example the observer pattern is presented, the two functions *notify()* and *update()* have to be inserted in *SewingMachine* because the interface has to be notified about changes and then updates itself. Furthermore, the sewing machine has to save which interfaces are registered. Thus, there has to be a variable *interfaces*, which can also be inserted automatically as well as a *register()* and *unregister()*

method which can be used to manage the interfaces. Next, the assignment relation is followed. Because the function at the end of it is just a parent function, all its sub functions are to be modeled next. Therefore, the three methods *changeStitchform*, *changeStitchlength* and *changeStitchwidth* are added to *SewingMachine*. The last component of the model is the Data object *Stitch*. This adds a third class to the class diagram. Unlike described by Rumpe[5], no association direction is inserted because of the fact that the direction is not identifiable in ArchiMate. Because the data object can be accessed from *Change stitchproperties*, it can be assumed that all of its sub functions can store information in it. Thus, three properties are added and named after their corresponding application function. To give a reasonable name, the verb is removed. Because the direction of the access relation goes from the application functions to the data object, the application functions have writing access to it and setters are added to all three properties. Getters are added as well because *SewingMachine* has reading access to the data object. The result is shown in figure 2.

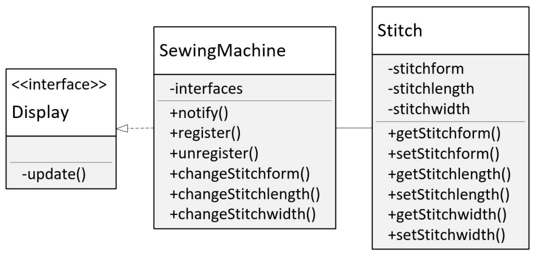


Figure 2: Sewing machine example as UML class diagram

Next, the model is transformed into a sequence diagram using the steps described in section 3.2. The resulting sequence diagram is shown in figure 3. Three classes were created in the class diagram and analogously three objects are created in the sequence diagram: the application component *SewingMachine*, the data object *Stitch* and the interface *Display*. Because an interface is offered, a user has to be added who triggers the interface which then creates a new activity box in *SewingMachine*. No concrete objects are expresses so the class name is used with a preceded colon. In this example three application functions are modeled which are triggered one after another. Thus, one single sequence diagram is created. The first action to take place is a change of the stitch form. because *changeStitchform* can access the data object, a call to *Stitch* where the new value of the changed stitch form is saved, is sketched. After the change of the stitch form is executed, the change of the stitch length and width are triggered. In the sequence diagram this is expresses by adding the two actions just behind *changeStitchform* at the same activity box of *SewingMachine*. Both can access *Stitch* so both update the corresponding property in the data object using the offered setter. At last, the interface *Display* is notified and updates itself. Adding the notification of the interface at the end of this sequence is done intuitively but cannot be derived from ArchiMate. In the ArchiMate model there exists no order of actions (except from the trigger relation which cannot be applied here) such that the order has to be scheduled within the transformation process (as described above). At this point, the reading access of *SewingMachine* to the data object *Stitch* cannot be modeled because there is no information available on which point *SewingMachine* is accessing *Stitch*. Thus, this access relation is left out in the sequence diagram.

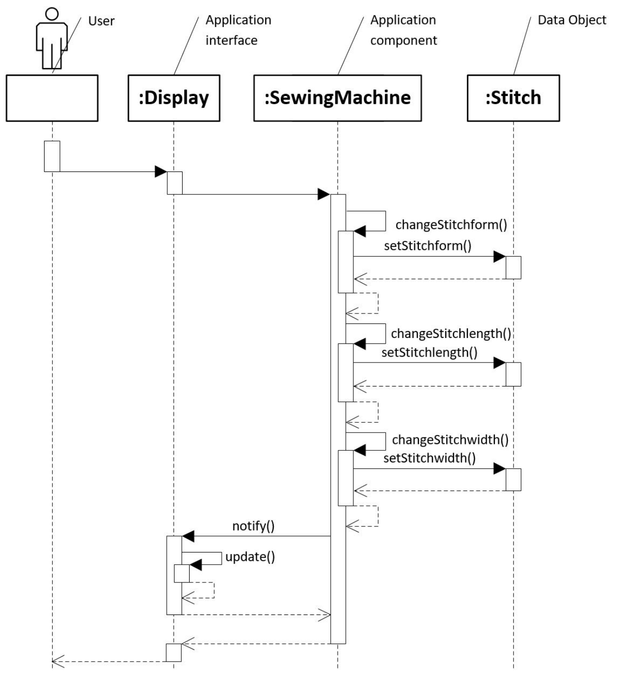


Figure 3: Sewing machine example as UML sequence diagram

5. DISCUSSION

Because there sometimes is a lack of information in the ArchiMate model, the diagrams cannot be created completely. For example, the temporal and causal context sometimes cannot be displayed adequate or data types (for example for returns of functions/methods, for parameters of functions or for properties and variables) as well as access privileges are not available. But developers usually want the ArchiMate model to be as complete as possible. Thus, they may think about using additional ways to store this data in order to create a more detailed model suitable for every user[11]. In the following, three notations to do so are to be mentioned: CMDB, ER-diagrams and UML. Because they are used to store additional information besides the model they can also be used to gain the lacking information which are needed for a more precise class and/or sequence diagram.

CMDB

The configuration management database is one source for additional information. It consists of configuration items which can be any entity of an IT component, their attributes, and their relations to each other[3]. Hence, this type of database can provide additional information about how the components are connected. It may give a causal context which is important for a valuable UML diagram. There may also be some information about the data used in the application and where they are stored. This can for example be helpful in the sequence diagram, where calls to data objects can be refined, or to identify data types for the class diagram.

ER-diagrams

An Entity-Relationship diagram can give insights into certain data of for example an existing database. This type of diagram models entities, relationships, and properties. Entities are the primary things, e.g. humans, things, or concepts, about which data should be collected. Relationships express the relation between entities and attributes model the properties of entities or relationships[7]. Because it models data together with relations and further information, it can be used to describe an ArchiMate model more detailed. For example, the properties of a data object can be described and how they are connected to other elements of the diagram. This information can be used to generate more appropriate UML diagrams.

UML

As described by The Open Group[11], which develops ArchiMate, UML can be used to convey deeper understanding of the modeled application. The combination of both languages allows to generate an extensive model of the business including detailed information of used applications and technology. It is described that specific nodes can be linked using UML deployment, component or activity diagrams. The type of diagram depends on the information that are to be displayed additionally.

If, for example, an activity diagram is available, the sequence diagram can be refined, because the activity diagram outlines the flow of an application including causal and temporal dependencies. These dependencies can be adopted in the sequence diagram. Hence, the combination of an ArchiMate model and a corresponding activity diagram can ease the process of generating a sequence diagram for this part of the application.

6. CONCLUSIONS

As shown in the previous sections, it is, with restrictions possible to transform ArchiMate models into UML class diagrams and UML sequence diagrams. Some components can be transformed directly while others need further processing. Because of the lack of information, especially when creating a sequence diagram or a distinct class diagram, additional information may be required, if a correct and complete diagram is needed. The three sources mentioned above may fill this gap. But because the semantics of these sources is not consistent, it is not possible to create a certain scheme at this point. It highly depends on the realization of these models if and how they can be used to refine the diagrams generated from ArchiMate. To construct a guideline wherewith it is possible to universally take these additional sources into account, has to be done in future.

This leads to two possible options if an organization wants to use a guideline like the one described in this paper: First, if there are more than one possibilities to transform an ArchiMate model to a UML diagram, the user has to decide manually which one is to be used. Furthermore, if there are information missing to model the diagram, the user has to manually insert this information. Second, the organization strictly uses a database with a predefined structure and alters the guideline in a way that it always looks up missing fragments at a certain point in this database.

7. FUTURE WORK

There also is a need for future research for the two layers which were not regarded in this paper: the business (beyond what Wiering[12] already described) and the technical layer. It is to be checked whether these layers contain information which can be used in an UML diagram of the application layer and thus can be used as additional source additionally to the sources mentioned in section 5. Moreover, an approach for an automatic realization of the transformations described in this paper could to be developed. It may be possible to use already existing logical structures for code generation from UML diagrams and adjust them, for example adapting finite state machines as described in section 2. Another possibility is not to see the generated UML diagrams as a standalone template for software engineers but as an extension to the existing ArchiMate diagrams, as The Open Group[11] already described. Thus, the generated diagrams can be used as basis for further development and serve as reference work if someone is interested in deeper understanding of ongoing processes. Hence, the diagrams need not to be complete, because they are only used as second source after the ArchiMate model.

Another field of future work is the synchronization between the ArchiMate model and the generated UML model. For example, if software engineers change the UML diagrams, these changes have to be put back into ArchiMate.

8. REFERENCES

[1] C. Braun and R. Winter. Integration of it service management into enterprise architecture. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1215–1219, New York, 2007. ACM.

[2] H.-E. Eriksson and M. Penker. *Business modeling with UML: Business patterns at work*. John Wiley & Sons, New York, 2000.

[3] H. Madduri, S. S. B. Shi, R. Baker, N. Ayachitula, L. Shwartz, M. Surendra, C. Corley, M. Benantar, and S. Patel. A configuration management database architecture in support of ibm service management. *IBM Systems Journal*, 46(3):441–457, 2007.

[4] M. Rincon-Nigro, J. Aguilar, and F. Hidrobo. Generación automática de código a partir de máquinas de estado finito. *Computacion y Sistemas*, 14(4):405–421, June 2011.

[5] B. Rumpe. *Modellierung mit UML: Sprache, Konzepte und Methodik*. Xpert.press. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2011.

[6] J. Sejans and O. Nikiforova. Problems and perspectives of code generation from uml class diagram. *Scientific Journal of Riga Technical University. Computer Sciences*, 44(1), 2011.

[7] I.-Y. Song and K. Froehlich. Entity-relationship modeling. *IEEE Potentials*, 13(5):29–34, 1995.

[8] The Open Group. Welcome to togaf^{TM}: The open group architecture framework: Version 8.1.1, 2006.

[9] The Open Group. *TOGAF Version 9.1*. TOGAF series. Van Haren Publishing, Zaltbommel, 1st edition, 2011.

[10] The Open Group. Archimate® 2.0: A pocket guide, 2012.

[11] The Open Group. Using archimate® language with uml®, 2013.

[12] M. J. Wiering, M. M. Bonsangue, R. van Buuren, L. Groenewegen, H. Jonkers, and M. M. Lankhorst. Investigating the mapping of an enterprise description language into uml 2.0. *Electronic Notes in Theoretical Computer Science*, 101:155–179, 2004.

Current state of best practices for developing automated software tests

Joel Hermanns
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
joel.hermanns@rwth-aachen.de

Horst Lichter
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
horst.lichter@swc.rwth-aachen.de

ABSTRACT

In large and complex software projects testing can quickly become cumbersome and painful. Therefore automated testing is needed. In the beginning of a project this approach requires more effort but this will - if done correctly - pay off later on by helping to keep the quality high.

Automated tests are either implemented using a domain specific language and a corresponding test framework or using a general purpose language, such as Java or C++. In the latter case the development of the tests is a software project itself and as such most of the best practices for developing software also apply here. For example, we want the code of the tests to be easily understandable and maintainable.

While there is plenty of literature about designing, developing and refactoring software this is not the case for automated tests. This paper aims to discuss existing literature about design patterns and best practices for developing and refactoring automated tests as well as missing one.

To group the literature according to the tests we will develop so-called influencing factors which are properties of the tests or the system under test and influence which best practices apply in this case. The existing and missing best practices will then be discussed broken down into these influencing factors.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

automated testing, continuous integration, continuous delivery, software testing, software quality

1. INTRODUCTION

Software is playing an important role in many domains nowadays. To guarantee the software is working correctly testing is mandatory. When the software project reaches a decent size manually testing all possible scenarios can become really cumbersome and time consuming. In this case automated testing might be a good option if not the only option. Automated testing means the test execution is fully automated, i.e. all required steps can be done by a computer, in difference to manual testing where most steps are performed by a human (for a more detailed definition of automated software tests see [20]). This makes it way easier and cheaper to repeatedly test the software, e.g. on a daily or weekly basis. Supporters of continuous integration go as far as suggesting all developers should integrate their changes early and often and tests should be run for every single change [16].

However, since automated tests are meant to be executed by a computer they need to be implemented in a form the computer understands. At first glance one might say that the cost for automated testing is higher than that of manual testing, given the implementation cost of the former. But since automated tests can help to avoid regressions and reintroduction of bugs later on, the quality of the software project can be kept high and therefore the effort for implementing these tests pays off if done correctly [26, 21].

There is plenty of literature about the successful implementation of automated testing [23, 18, 24, 25]. However, as Kaner and Berner point out the implementation can also easily go wrong consuming more time than it is meant to save [22, 12].

To avoid making the implementation a failure we are interested in implementing good tests, i.e. tests with certain properties. Properties of a good test include but are not limited to [26]:

- **Validity** - Obviously the most important property. A test should always be valid so in case it fails this is a sign that something is wrong with the system under test.

- **Determinism and repeatability** - If nothing has changed the result of a test is always the same. This is important if we want to debug why a test fails.

- **Independence** - This means a test should not lead to different results if run in parallel with other tests. So it should not interact with the other tests in any way.

The Cambridge Dictionary defines *best practice* as “a working method or set of working methods that is officially accepted as being the best to use in a particular business or industry, usually described formally and in detail” [3]. So in case of implementing automated test best practices are methods or ways to help us reach the properties described above and avoid common mistakes and problems that may arise when using automated tests.

Since automated tests need to be implemented in code there are basically two options how this can be done, either by using a DSL and a corresponding test framework or a general purpose language, such as Java or C++. In the former case a language is used that is focused on writing tests and therefore typically does not allow to do much more. In this case the best practices are directly coded into the test framework and the implementation of the language itself. In the latter case, the use of a general purpose language, the implementation of automated tests becomes a regular software project itself and as such we are also interested in maintainability, complexity, efficiency and realiability as well as robustness and extensibility of the tests itself.

The remainder of this paper is structured as follows. In chapter 2 we will introduce so-called influecing factors of automated tests. These are properties of the test or the system under test we extracted which influence which best practices are applicable. Broken down into these different factors chapter 3 aims to describe and discuss existing literature about best practices for automated software tests. Followed by chapter 4 in which we point out the missing best practices and discuss what we have seen and what we can learn from it. The paper will be finalized with a conclusion and ideas for future work.

2. INFLUENCING FACTORS OF AUTOMATED TESTS

In this section we will introduce influencing factors of automated tests. So we need to clarify what an automated test is. In this paper, we call a test automated if all steps required to run the test are automated. It should be as simple as pressing a button to run the test. The computer will execute all required steps and emit the result which is either the test passed or it failed.

However, this does not include the setup of the environment, which is a completely different aspect. Generally, we assume the environment meets the requirements to run the test when the test is started. Nevertheless, if the execution of the tests requires further steps to be done manually every time we want to run the test it is not fully automated and therefore cannot be called an automated test.

When implementing such automated test there are several aspects one needs to care about. So we are interested in the best practices for doing so. This section comprises important influencing factors with regard to best practices. The grouping was done by analysing how different properties of the test, the system under test or maybe the environment influence the design, development, setup and refactoring of automated tests, especially in terms of best practices.

The first three groups we will talk about describe factors that come with distinct manifestations we can differentiate. Whereas the last group is just a listing of additional properties that influence the best practices but for which we do not need to distinguish between different manifestations because they are not of interest.

2.1 Level of testing

The first group of factors we want to introduce is the level of testing. In case of automated testing it is sufficient to distinguish between two levels of testing as we will see later.

- The first level is what we will refer to as **unit level**. This level includes all kinds of tests that examine just small parts of the whole software. As the name suggests this includes of course unit test, which are tests for just a single function or a class. Additionally, it includes module and/or component test. Generally, these are tests that focus on a large part of the software than unit tests, e.g. a module, component or package [21]. And finally the third kind of tests included in this level are integration or contract tests, which are meant to test if several components or modules integrate with each other. An important property of these kinds of tests we can observes is that they do not require the software to run as a whole. Usually, every other unit or dependecy except for the unit under test is mocked away [26].

- The second level we want to discuss here is what we call the **systems level**. This level covers all kinds of test that have a higher level view on the software than the tests in the unit level. Basically, this means the system is tested as a whole which is why it is called systems level. Usually, this includes acceptance or customer testing as well as other forms of systems testing, such as end-to-end testing [14, 21]

2.2 Types of tests

The second group we want to discuss here are the types of tests. We split this group into the following two manifestations:

- **Functional** tests focus purely on business logic. The test will either test a whole feature or just parts of the implementation depending on the level. As said earlier an automated test can always either pass or fail. In case of funcational tests this is very simple to achieve as testing certain scenarios always have a precise correct result. So checking if this was the result we got and failing accordingly is pretty simple.

- **Non-functional** on the other hand are tests for non-functional requirements such as performance or security concerns. In contrast to functional tests these often require specialized tooling to make automation feasible [12]. For example, tooling to simulate a large load to examine how the system performs under heavy load or tooling to simplify penetration testing of software by making use of existing tests (c.f. metasploit [6]).

2.3 Environment

The third influencing factor we want to discuss here is the environment the tests are run in.

In the case of automated tests it is sufficient to distinguish between the following two environments:

- **Development machine**: The computer on which the development is done.

- **Specialized test environments**, e.g. a test setup, or a deployment pipeline as part of a continuous delivery implementation.

It is important that the environment the tests are run in fullfills certain requirements, e.g. running a certain database.

2.4 Technology

This group of factors comprises various kinds of properties of the software system under test and its domain. We will not identify discrete, distinct manifestations here but rather introduce aspects that are worth thinking about when implementing the automated tests and influence the relevant best practices as we will see in the next chapter.

- The first aspects are **language, framework, libraries and tools** used for writing the tests as well as for writing the actual software system. Obviously, the language and framework used for writing the tests makes a difference for which best practices are relevant. However, this does also apply for the software that is tested itself. We will discuss examples in more detail in the next chapter.

- **Access to the system** the software is meant to run on eventually is the second aspect. For example, we need to employ a different strategy for testing a software that is targeted to be running on the developer machine than we need to employ for a software running on microcontroller which is typically cross-compiled and flashed on the target system.

- The final aspect we discuss here is the **software architecture** of the system under test. For example, one can easily imagine that the best practices for a testing strategy for standalone monolithic application, like a browser running on a desktop computer is different from the one for a distributed system, like a highly-scalable database of servers.

2.5 Summary

In the preceding sections we gave a brief overview of the important influencing factors we developed. Different groups and their manifestations were discussed and their influence on the relevant best practices was briefly mentioned. In the next section these factors will be address by reviewing existing literature.

3. EXISTING BEST PRACTICES

In the following we are going to introduce existing literature on best practices for developing automated software tests. The discussion will be based on the grouping of influencing factors introduced in the previous section. For better understanding we will introduce an example application. For this example application we will go through different test scenarios, explain what the test scenario means in context of the example application and analyze the existing best practices.

3.1 The sample system under test

As an example application we will consider a fictonal web-based shop system. For this paper we assume the system is written in Java. The shop allows different users to log in. Users with the role of an admin or an owner can modify the product catalog, i.e. add, remove and update products and their details, e.g. pricing. Other users can add products to their cart and perform a checkout.

Other details are not relevant here as the example is just for illustration purposes.

3.2 Functional testing on unit level

The first test scenario we are looking at is testing on unit level. In case of our example application this refers to testing single classes or functions. Since our web-shop is written in Java the language for writing the tests will also be Java as the test code uses mostly internal private APIs.

We care about best practices that apply on code level meaning best practices on how to structure, design and refactor the code used for the automated tests.

Meszaros gives a comprehensive reference of useful design patterns that help when refactoring test code [26]. He explains various patterns in greater detail and refers to them as “xUnit test patterns”. The list was developer over a larger period of time by collecting patterns he found useful himself when implementing tests or others described as useful ones.

Not all patterns are as relevant for testing on unit level as others. One example pattern that is particulary interesting for unit testing is the mock object pattern. “A Mock object is an object that replaces a real component on which the [system under test] depends so that the test can verify its indirect outputs” [26]. On unit testing level we want to isolate the unit we test as far as possible so we can focus the testing on just the unit and can dictate the conditions.

Clifton and Truyers describe additional patterns [15, 30]. For example, Truyers shows how to decouple the constructor of a domain from the actual construction in the tests of this domain class using the builder pattern. So if the constructor is changed only the builder implementation needs to be changed which vastly reduces the maintainence overhead.

The mentioned resources discuss test design patterns in general. If we consider our example application we probably want to use a web framework to simplify the implementation. Usually this comes with its own set of best practices specifically for the framework. For example, if we use the Spring Framework [8] we can find a guide specifically for unit and integration testing in applications using the framework [9]. The same goes for popular frameworks in other languages, such as Ruby on Rails or Python’s Django [5, 10].

One limitation of the best practices discussed so far is that they only apply when using object oriented languages, as we do in the case of our example application. However, we are also interested in best practices when using languages which follow other paradigms. For some of the patterns we can find equivalent ones in other languages. For example, if we reconsider the mock object example from above we can find an equivalent implementation based on the cmocka framework for C code as explained by Schneider et. al. [2, 29]. Similarly there are patterns for newer languages that are not object oriented, such as Go [4]. Hashimoto describes a set of advanced testing techniques for Go [19]. One commonly used pattern in automated tests for Go code is the so-called table driven test. The basic idea of this pattern is quite similar to the pattern “Template” introduced by Rybalov [27]. The idea behind the template pattern is to move common test logic into an abstract super class such that the actual test cases only need to implement certain hooks in a subclass. The following shows a possible implementation of the super class:

```
1  public abstract class ShoppingCartTemplate extends TestCase {
2      public ShoppingCartTemplate(String pName) { super(pName); }
3      protected void setUp() {
4          login();
5          searchForItem1();
6          cartAction1(); // <—— test hook 1
7          searchForItem2();
8          cartAction2(); //<——— test hook 2
9          logout();
10     }
11     //Subclasses are supposed to implement these test hooks anyway they like.
12     abstract protected void cartAction1();
13     abstract protected void cartAction2();
14     //test tool specific implementations of methods
15     //(e.g. login(), searchForItem1(), etc.)
16 }
```

The *setUp* function in lines 5 to 12 contains the common test logic. It first performs common setup logic, such as search and login. Afterwards it calls two abstract functions *cartAction1* and *cartAction2* in line 7 and 10. These abstract functions can easily be overwritten in subclasses and as such allow to reduce the effort to implement new tests using the same basic test logic.

In Go one typically makes use of anonymous structs. The following code shows a possible implementation:

```
1  import "testing"
2
3  func TestShoppingCart(t *testing.T) {
4
5          testCases := []struct {
6                  action1 func()
7                  action2 func()
8          }{
9                  {
10                         // 1st testcase
11                         action1: func() {...},
12                         action2: func() {...},
13                 },
14                 {
15                         // 2nd testcase
16                         action1: func() {...},
17                         action2: func() {...},
18                 },
19                 // additional testcases go here...
20         }
21
22         for _, testCase := range testCases {
23                 login()
24                 searchForItem1()
25                 testCase.action1()
26                 searchForItem2()
27                 testCase.action2()
28                 logout()
29         }
30 }
```

The struct defined in lines 5 to 8 contains two functions implementing the cart actions that were abstract members of the class in the Java example. The common test logic can be found in lines 29 to 34. It is called for every testcase by iterating through the array of structs and calling the corresponding functions in line 31 and 33.

So we can see that the same idea can be expressed using very different constructs in two very different languages.

In general, we can say that every kind of framework or language which has reached a decent maturity proposes its own set of best practices for writing tests on unit level.

Because on unit level the system is tested in small portions, the tests are in general independent of the environment as an influencing factor. The only expection we can find here is in case the system under test and the tests get cross-compiled and do not run on the development machine, e.g. in development for embedded devices. In this case the compilation result needs to be brought to the target device before execution.

3.3 Non-functional testing on unit level

In the preceding section we talked about functional testing on unit level. Now we want to analyse the case for non-functional testing. With regard to our example application we would, for instance, want to benchmark how a method for calculating the total value of the shopping cart performs. Benchmarking, i.e. applying performance testing, on very small units such as functions is called micro benchmarking. For writing such benchmarks it is recommended to use specialized tooling. Jmh is such a tool for Java based application [7] and would be a potential candidate for our web-shop.

One aspect one needs to think about when adding such tests to an automated test suite is how to evaluate the results of such tests. The results of benchmarks are heavily influenced by the environment they are running in. Therefore one needs to provide an environment independent, automated evaluation schema of the benchmark results. An option is to leave the analysis of the result to a human but this will require manual steps for every execution of the test.

3.4 Functional testing on systems level

In the following we look at another possible test scenario, namely functional testing on systems level. When we look at the fictional web-shop again possible tests include the following user stories:

- Adding an item to the cart (customer test)
- Performing a checkout (customer test)
- Updating payment information (customer test)
- The complete flow from first visit, login to checkout (end-to-end test)

Most of the best practices discussed in the previous test scenario also apply here except that there is a shift of the importance of the design patterns to be used. For example, the mock object pattern is not interesting for testing on systems level as the system should be tested as a whole without parts being mocked. Rybalov explicitly describes test design patterns for customer testing specifically [27]. For example, he describes the "Template" pattern we have discussed in the previous section already. This is especially interesting for systems level testing as tests in this case often require more complex initialization and teardown procedures which will repeat for every test case.

In addition, the influence of the environment as a factor becomes more critical. Now that the whole application needs to be run we have more requirements for the environment. Depending on the technology used and the architecture of the software the requirements for the environment and the setup are more complex. For example, the fictional web shop should be setup on its own isolated from the tests. It will probably require a database server and the tests will communicate with the web-shop via a network connection. So when developing the automated tests this will influence how the tests are implemented, especially with regard to setting up and configuring the application.

Finally, we can also find a difference in applicable best practices with regard to architecture of a system under test. One example are microservice based systems. In a microservice architecture the whole system is composed of a larger set of services each implementing a small part, i.e. functionality, of the overall system. Due to the sheer number of different services it is often not feasible to set up the whole system when running system tests. This is where the testing strategy changes. A new version is usually tested by deploying it and exposing it to a small part of the whole user traffic [13, 28].

3.5 Non-functional testing on systems level

The final scenario we want to discuss is non-functional testing on systems level. Exemplary test scenarios for the example online shop are

- Make load tests, i.e. check how many concurrent users the system supports
- Measure performance of database queries
- Make penetration testing to find security problems

As Berner et. al. state these forms of testing heavily rely on proper automation and tool support [12]. Typically the tests do not require much variance regarding the test logic but rather variance in the data that is used as input for the system. Therefore non-funcational testing often follows the data driven tests pattern where the input data is read from a file [26]. This allows to quickly reach great coverage without needing to write a lot of test logic.

As such specific tools and frameworks are used usually to implement non-funcational testing on systems level. For instance, Gavrila et. al. describe a framework for automated performance testing of HbbTV streaming solutions which uses XML based data input for the tests [17].

3.6 Summary

In the preceding section we have looked into existing best practices for implementing automated software tests. To do so we discussed four test scenarios for an example system under test based on the influencing factors introduced in 2

4. MISSING BEST PRACTICES

In the following we want to look into cases where no best practices exist for in literature. Therefore we want to discuss two examples from our own experience that involve testing on embedded devices:

- The first example is the development of a LED controlling software running on a microcontroller. The microcontroller allows communcation over a serial communcation interface, such as SPI or I2C, using a custom protocol. During development the software is cross-compiled on the development machine and then flashed on the device. To test the protocol and the controller implementation test commands need to be sent over the serial interface and the response needs to be verified.

- The second example involves the development of an operating system for an embedded device. Similar to the first example the development is done on a different machine where an image is built from which the embedded device can boot. When making changes to the implementation we need to make sure the changes have the desired effect and we did not introduce a regression. So for instance, we check if the system still boots correctly, if all parts of the system are configured correctly as well as if there are no security problems.

What both cases have in common is a limited access to the system, we cannot easily run tests on the same system and we are not able to run the code itself on the development machine. Furthermore, the software runs on a completely different hardware from the machine the software is developed on. In general, this is also true for the development of mobile applications for devices like the iPhone or Android smartphones. In both cases it is technically possible to write automated tests. While we cannot find literature describing best practices for writing automated tests for the former one, support for writing tests is already included in the respective platform in the latter case [11, 1]. So to summarize the embedded ecosystem still lacks detailed best practices and tools to implement automated test suites.

5. CONCLUSION AND FUTURE WORK

In this paper we have analyzed literature about best practices for implementing automated testing. To do so we have introduced several influencing factors of tests. Based on the extracted factors we have looked into existing literature as well as pointed out missing literature. We have discussed potential test scenarios for a fictional web-base online shop and analyzed which best practices are applicable and might help implementing an automated testing strategy. This was done on unit level as well as system level both for functional and non-functional tests.

We have briefly discussed design patterns that simplify implementation and maintenance of tests and pointed out the existing literature that describes these patterns in more detail.

In section 4 we analyzed two examples for which we could not find existing best practices and explained the problems.

In the end we have seen that the tooling and framework support is crucial. For applications similar to our ficional web-shop we can find plenty of tools that help to write automated tests and propose their set of best practices. The result is that the developer can focus on writing the actual test logic and does not have to worry about other details such as test execution and discovery.

While there are many best practices that help to implement the test logic and are applicable to all different kinds of technology the missing best practices described in section 4 are all related to tooling support.

As such possible future work can focus especially on the investigation of missing tooling and framework support for embedded devices as well as an improvement for mobile devices.

6. REFERENCES

[1] Android studio testing. https://developer.android.com/studio/test/index.html. Accessed: 2016-11-02.

[2] cmocka. https://cmocka.org/. Accessed: 2016-11-02.

[3] Definition best practice. http://dictionary.cambridge.org/dictionary/english/best-practice. Accessed: 2016-11-02.

[4] Go website. https://golang.org/. Accessed: 2016-11-02.

[5] A guide to testing rails applications. http://guides.rubyonrails.org/testing.html. Accessed: 2016-11-02.

[6] Metasploit website. https://www.metasploit.com/. Accessed: 2016-11-02.

[7] Openjdk - code tools: jmh. http://openjdk.java.net/projects/code-tools/jmh/. Accessed: 2016-11-02.

[8] Spring framework. https://spring.io. Accessed: 2016-11-02.

[9] Spring framework testing. https://docs.spring.io/spring/docs/current/spring-framework-reference/html/testing.html. Accessed: 2016-11-02.

[10] Testing in django. https://docs.djangoproject.com/en/1.10/topics/testing/. Accessed: 2016-11-02.

[11] Testing with xcode. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/04-writing_tests.html. Accessed: 2016-11-02.

[12] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*, pages 571–579. ACM, 2005.

[13] S. Bruckner. Microservices at gutefrage.net - part 2 - continuous integration and deployment. https://medium.com/gutefrage-net-engineering/microservices-at-gutefrage-net-part-2-continuous-integration-and-deployment-6c5b97f40245#.2wgidufil. Accessed: 2016-11-02.

[14] T. Clemson. Testing strategies in a microservice architecture. http://martinfowler.com/articles/microservice-testing/, 2014. Accessed: 2016-11-02.

[15] M. Clifton. Advanced unit test, part v - unit test patterns. https://www.codeproject.com/articles/5772/advanced-unit-test-part-v-unit-test-patterns, 2004. Accessed: 2016-11-02.

[16] P. M. Duvall. *Continuous Integration*. Pearson Education India, 2007.

[17] C. Gavrilă and C.-Z. Kertész. Automated performance testing of end-to-end streaming solutions over hbbtv architecture.

[18] J. Gmeiner, R. Ramler, and J. Haslinger. Automated testing in the continuous delivery pipeline: A case study of an online company. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–6. IEEE, 2015.

[19] M. Hashimoto. Advanced testing with go. https://speakerdeck.com/mitchellh/advanced-testing-with-go, 2016. Accessed: 2016-11-02.

[20] D. Hoffman. Test automation architectures: planning for test automation. In *Quality Week*, pages 37–45, 1999.

[21] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[22] C. Kaner. Architectures of test automation. *STAR West*, pages 1–17, 2000.

[23] Z. Liu and P. Mei. Automated testing for large-scale critical software systems. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*, pages 200–203. IEEE, 2014.

[24] J. Lu, Z. Yang, and J. Qian. Implementation of continuous integration and automated testing in software development of smart grid scheduling support system. In *Power System Technology (POWERCON), 2014 International Conference on*, pages 2441–2446. IEEE, 2014.

[25] X. Ma, N. Wang, P. Xie, J. Zhou, X. Zhang, and C. Fang. An automated testing platform for mobile applications. In *Software Quality, Reliability and Security Companion (QRS-C), 2016 IEEE International Conference on*, pages 159–162. IEEE, 2016.

[26] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[27] M. Rybalov. Design patterns for customer testing. *Automated Testing Guy. Disponível em http://www.autotestguy.com*, 2004.

[28] B. Schmaus. Deploying the netflix api. http://techblog.netflix.com/2013/08/deploying-netflix-api.html. Accessed: 2016-11-02.

[29] A. Schneider and J. Hrozek. Unit testing with mock objects in c. https://lwn.net/Articles/558106/. Accessed: 2016-11-02.

[30] K. Truyers. Flexible and expressive unit tests with the builder pattern. https://www.kenneth-truyers.net/2013/07/15/flexible-and-expressive-unit-tests-with-the-builder-pattern/, 2013. Accessed: 2016-11-02.

Modelling Architectural Complexity: An Overview

David Duong
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
david.duong@rwth-aachen.de

Horst Lichter
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
horst.lichter@swc.rwth-aachen.de

ABSTRACT

Software architecture is defined as the organizational structure of a software system. It contains all design decisions during the development process. As one of its properties, architectural complexity has a strong correlation to many non-functional qualities i.e. maintainability and extendability. To this day, there is no commonly agreed upon definition of architectural complexity. The correlation between cohesion, coupling and the complexity is, however, widely accepted. Therefore, many structure-based metrics were introduced over the last years, especially cohesion and coupling metrics.

This paper aims to present an overview about architectural complexity. We first present the definitions available in the literature. Since, architectural complexity can be accessed by structure-based metrics sets, we then focus on the classification of structure-based metrics according to the approach types, measured artefacts as well as measured software attributes. The classification, however, is not complete, we will discuss about the limitations of the current state. Furthermore, we present a formal model of architectural complexity SACM based on the cognitive science. At the end, we conclude that there is a need of research on new metrics.

Keywords

Software Architecture, Architectural Understanding, Architectural Metric, Complexity Model

1. INTRODUCTION

Software architecture depicts different high level views on the developing software system. Thus, the participants of a given project have the same "big picture" of the product, i.e. higher level organizational components in place of fine-grained implementation artefacts. The key role as well as the quality of software architecture is generally recognized, since "architectures allow or preclude nearly all of the system's quality attributes"[8].

Complexity has always been one of the most important topics in software engineering. Talking about complexity in the architecture domain, cohesion and coupling between and within the components always come to mind. Since cohesion and coupling are widely accepted attributes of a software architecture, there are only few studies about "architectural complexity". Therefore, the scientific definition of architectural complexity hasn't established yet. There could be other factors related to architectural complexity than coupling and cohesion, which have not been identified yet.

In the last decades, numerous attempts have been made to provide some measure of software architecture, especially coupling and cohesion. However, an overall quality of the architecture can not be achieved. Besides, there is also research on evaluation methods to evaluate the quality of a software architecture. Together with architecture metrics, a model for architectural quality can be defined[7].

In this paper, we will first introduce the definitions of architectural complexity available in the literature in section 2. From Stevanetic *et al.*'s assessment of structured-based metrics, we will present their classification on metrics measuring architectural understandability in section 3 and discuss the limitations. A formal complexity model based on cognitive science will be presented in section 4. Finally, section 5 concludes this paper.

2. DEFINITION

The definitions of "architecture" and "complexity" have been standardized in "IEEE Vocabulary for Software Engineering"[1]. In 1990s, the term "software architecture" became popular. It was defined in several works, such as in [4, 20]. The first formal standard for software architecture was introduced in year 2000 and in 2011. The term "architecture" is defined as: "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment"[1]. By definition, the software architecture contains information about all functional and non-functional requirements of the system. In practice, such enormous information is divided with the aid of views and their corresponding models. Furthermore, a software architecture contains also "the principles guiding its design and evolution"[1]. All design decisions and principles effect the architecture, especially its complexity, either positively or negatively[12].

The proper definition of the term "complexity" depends on the domain. In algorithm complexity theory, the algorithms are analyzed according to their efficiency in form of O-notation(space and time complexity). The cyclomatic complexity defines the number of linearly independent paths within a source code. The number indicates the difficulty to test the given source code. In the software architecture domain, there are different opinions on the definition of complexity based on the size of the system, how complicated the system is or how decomposable a system is.

In "IEEE Standard Glossary of Software Engineering Terminology", the term "complexity" is defined as "the degree to which a system's design or code is difficult to understand because of numerous components or relationships among components"[1]. It means that the complexity of a system consists of the complexity of the system's design and implementation, which are found in different views and source code. According to this definition, the software complexity implies the architectural complexity, since the architecture is design part of the software. In addition, the architectural complexity is related to the number of components and relationships among them. Similarly, Yingxu Wang defines that "the architectural complexity[...] is determined by the number of data objects at system and component levels"[25]. Moreover, in Wang's opinion, the architectural complexity is also a part of the overall cognitive complexity of a software system next to other complexities, such as symbolic, relational, operational and functional complexities.

IEEE also defines that the architectural complexity can be accessed using any set of structure-based metrics[1]. Aside from the numerous components or relationships among them, the degrees of cohesion and coupling within and between the components are widely accepted as factors affecting the architectural complexity. For instance, those factors were considered by Sangwan *et al.* during the complexity analysis of a software system[21]. They stated that the coupling between architectural units should be as low as possible to limit the change and error propagation. The software architecture can be presented as a dependency graph, which should be free of cycle. About cohesion, Sangwan *et al.* stated that the more cohesive a component is, the less complex it is; consequently, the less complex is the architecture considered to be.

Another approach on defining "architectural complexity" introduced by Lilienthal is based on a general approach that the complexity of a software system is divided into the complexity of a problem and the complexity of a solution[10]. Lilienthal then defines the architectural complexity as follow:

> *"the architectural complexity is the structural part of complexity of a solution, which is a result of design decisions of architectural elements and relationships among them"*[18].

The complexity of a solution emerges during the development process, while the architect is trying to design a functional solution for a problem domain. Moreover, architectural complexity can be found in implemented architecture and the mapping between designed architecture together with architectural style and the source code. The implemented architecture is more complex than the designed one, and it depends on the decision made by the architect on design and architecture style[18]. In addition, developers need to map the designed architecture to the implemented one of software system to verify their work. This is considered very complex task and it might become more difficult, if a drift between the designed and implemented architecture occurs. The architectural drift indicates the inconsistency of the designed architecture[12].

In summary, "architectural complexity" is a part of the overall complexity. It is the degree to which a system's design is difficult to understand. According to the definition in [1], the architectural complexity is related to different attributes, such as cohesion, coupling, and the numbers of components and relationship among them. Furthermore, the complexity can be assess by structure-based metrics.

3. ARCHITECTURAL METRICS

Measuring the complexity within a software system can be done in different ways. At the code level, the Halstead's difficulty and cyclomatic complexity by McCabe represent the common and quantitative metrics. At the design level, the description of a software architecture comprises multiple views, which offer lots of possibility on measuring architectural attributes of the system. Stevanetic *et al.* presented a systematic mapping study of software metrics[23]. Those metrics measure the attributes of a system at architecture level and according to the definition of IEEE, they also access its complexity. Stevanetic *et al.* only took the structure-based metrics measuring from the deployment view into account. This view captures mostly information on major functionalities of components and interaction between them. This section presents the classification of structured metrics introduced by Stevanetic *et al.* and its limitations.

3.1 Metric Classification

Stevanetic *et al.* classify the metrics according to their approach types, as well as the measured artefacts and attributes. There are three different approach types, i.e. internal structure based metrics, graph based metrics and specific model based metrics[23]. From our view, there are only two distinct approaches, i.e. direct and indirect metrics. The internal structure based metrics measures the understandability directly on the architecture of a given software. This approach type corresponds to the direct metrics. For instance, the metrics introduced by Hupta *et al.* measure coupling and cohesion of packages directly from the architecture without any transformation[13, 14]. The indirect metrics, however, need additional representation of a given architecture for the measurement. The simplest representation of a software architecture is the graph model. The metrics measure from graph models are classified as graph based metrics. For instance, Allen introduced his approaches based on information theory in [2], and also the later paper [3]. The relation between Allen's metric set and the architectural understanding was proved by Stevanetic *et al.*[22]. There are also specific representations of a software architecture, such as layered architecture models, composite architecture models. Metrics measuring from those representations are classified as specific model based metrics. Additional effort is needed for using those metrics, since the transformation to the specific representation is required.

According to the measured artefacts, Stevanetic *et al.* identified metrics measuring at high level structure as well as metrics related to single artefacts[23]. The measured software artefacts are architecture, component, module, package, component-to-component and graph node. As an artefact, the architecture takes into account every part of the system on different levels of the hierarchy. The high level artefacts are components and modules. However, those artefacts need a context to be defined properly. On the one hand, they might be on the highest level of the hierarchy. On the other hand, they might on lower level and contain lowest level artefacts, such as source code. Either way, component and package have well-defined functionality[23]. The low level artefacts are packages containing source code or other packages. Those packages builds a hierarchy on the lowest level of the architecture. Besides, the relations between component are classified as component-to-component artefacts. Since a graph model is the simplest representation of the software architecture and the architectural parts can be represented as graph nodes or edges, the graph node is the simples artefact.

According to the measured software attributes, the metrics are classified into different categories. The size metrics are quantitative metrics related to the number of elements inside the corresponding artefact, in part or the whole system. The coupling and cohesion metrics are well-known, widely accepted in the architecture community and already introduced in previous section. So far, those metrics measure basic software attributes, meaning they don't consider other metrics for computation. Other category of metrics is complexity metrics measuring the degree of connectivity between the units of a software system. The complexity metrics are, however, not basic metrics and consider the cohesion and coupling metrics. Similar to the complexity metrics, stability metrics take coupling metrics into account and measure the effort to take change on a artefact. In addition, there are quality metrics measuring from a specific representation of the software system. Those metrics needs the composite based software architecture for measurement and is based on the Multi-Attribute Utility Technique (MUAT)[23].

3.2 Limitations

By now, the classification of structured-based metrics is not complete, since only metrics measuring from the deployment view are taken into account. Stevanetic *et al.* mentioned an additional approach measuring on the meta-level of the architecture, such as OCL and UML[23]. One approach measuring UML was introduced by Lankford [17], which is not covered in the assessment by Stevanetic *et al.*. Another approach was introduced by Kazman *et al.*[16], which focuses on the reorganization of patterns implemented in a given software architecture.

For their work, Stevanetic *et al.* defined a maturity level assessment for the structure-based metrics. This assessment can be applied on the future research of structure-based metrics. It focus on different parameters, i.e. definition, mapping quality of the metrics to the external quality characteristics, level of validation, comparative analysis, usability, applicability and tool support. As a result, 80% of the metrics have a medium usability level; meaning, they require some effort in order to understand and compute the metrics. 72% of them require further improvements or more reasonable evaluation criteria in order to be applied in real projects[23]. The results show the potential deployment of those metrics in the development process. Besides, Stevanetic *et al.* identified the common problems reflecting "through ambiguities in metrics definitions, lack of empirical validation and lack of information [...] and missing comparative analysis [...]"[23]. In brief, further validations and works on those metrics are needed before the metrics can be applied into the development process.

4. ARCHITECTURAL COMPLEXITY MODELS

The number of architectural complexity models stands in contrast to the numbers of metrics. In the literatures, there are only few models for architectural complexity. For instance, Darcy *et al.*[9] created a structural complexity model of software based on Wood's task complexity model (see more in [26]) in order to prove their hypothesis. They considered software functionalities as tasks and those can be analyzed by Wood's model. With the aid of an adaption of Wood's model for the software domain, they defined cohesion and coupling concepts theoretically, validated the relation between them and structural complexity. Those concepts affect structural complexity in terms of software understanding and effort to perform maintenance tasks[9]. They claimed that the adaption of Wood's model can be applied on larger systems. The concept of cohesion and coupling can be applied throughout the whole software life for a better cost-benefit ratio.

In our research, we found another formal model for architectural complexity firstly introduced by Lilienthal[18] (complexity model of Lilienthal - CML). Similarly to the software quality model, the factor-criteria-metric-model (FCM-model) is used as framwork to design CML[19]. The CML is underlaid by cognitive science theory and general software engineering principles. However, there are limitations on CML, such as the focus of cognitive science theory only on a specific level of understanding, and the lack of validation on programming language[7]. Bouwers *et al.* introduced a new model based upon CML and inspired by system attributes of Bouwers **et al.**[6], the Software Architecture Complexity Model(SACM)[7]. However, this model was created in order to prove Darcy *et al.*' hypothesis and does not focus on accessing architectural understandability of a software system. Therefore, we won't take a closer look on this model.

In SACM shown in figure 1, the overall goal, the architectural complexity, is divided into five different factors specified in two categories, i.e. personal and environmental. The factors and criteria of the original CML cover most of the personal categories and are marked in gray color. The extension from CML to SACM is denoted in white color. In the next layer, there are criteria substantiating those factors, hence they are still abstract. The metrics for measurement are in the lowest level of the model.

4.1 Personal Factors

This category of factors focuses on the understanding of a single developer. During the development process, the developer must handle an enormous amount of information contained in a given software system and its architecture. Such information includes the components and relationships among them. Furthermore, as mentioned in section 2, the mapping rules between designed and implemented architecture is an essential and complex task of software developers. To answer the questions "How does a developer solve such a complex task", Lilienthal turned towards the field of cognitive science[18]. Within cognitive psychology, Lilienthal focuses on three mental strategies to deal with enormous information and complex mapping, i.e. chunking, formation of hierarchies and schema.

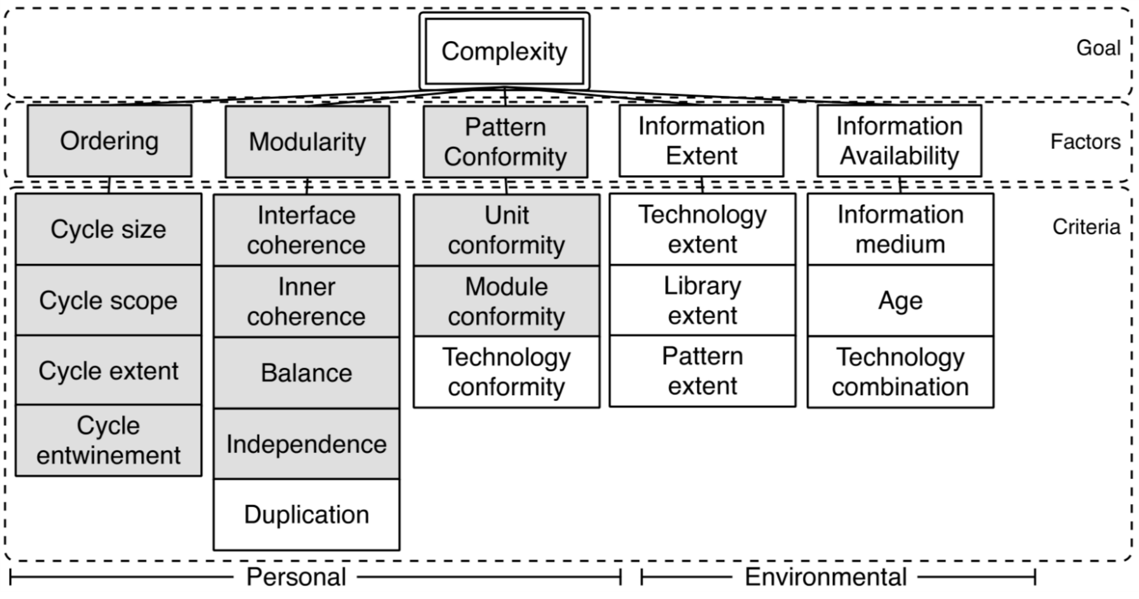


Figure 1: Software Architecture Cognitive Model(SACM)[7]

Chunking is a strategy focused on recoding small information units into a new larger one called a "chunk". By applying this strategy, human short-term memory is used in a more efficient way. Those chunks need to be a product of meaningful and cohesive information. This strategy can be found in the general design principles, such as modularization and abstraction through interfaces.

Another strategy is the formation of hierarchy, which is mostly used in combination with chunking. The information units are structured on different levels. As a result, the hierarchical structure allows developers to understand and process information in an easier way. In software construction, hierarchies are proven structures to reduce complexity. The design principles applied in this strategy are avoiding cyclic structures and layering, etc.

Schema comprise information units from the abstract and the real world. The abstract information units are the typical aspects and domain of derived instances of the real worlds. By applying in this strategy, design patterns are the result of using the excellent human ability to recognize and match pattern.

Based on these strategies in cognitive science and design principles in software engineering, Lilienthal defines the following three main factors in CML[18], which are classified as personal factors in SACM and have additional criteria[7].

- **Modularity** – checks the implemented architecture for decomposition. The components should be cohesive, encapsulate their behavior, and offer a cohesive interface.
- **Ordering** – checks the implemented architecture with regards to the directed, acyclic graph, which is build from the relationship between elements.
- **Pattern conformity** – looks for architecture drift using pattern matching and checks whether the mapping rules from the designed to implemented architecture are applied correctly.

4.2 Environmental Factors

Nowadays software systems, especially large-scale systems, evolve overtime. The amount of information contained in the architecture might become larger and exceed the capabilities of a single developer. In addition, the business requirements will also change, which leads to the adaption of the implemented architecture. Depending on the available time for the adaption, developers need to work together in some form of distributed cognition. To handle the situation, one strategy is to lower the amount of information needed to understand the implemented architecture. However, the degree of difficulty to understand the information depends on the developer processing them.

All in all, the context for this information should be available, especially the implemented architecture. In addition, it needs to be kept to a minimum. Furthermore, the representation of the information might allow a better way to understand the architecture. Bouwers *et al.*[7] observed those needs of information and define the following environmental factors:

- **Information Extent** – checks the amount of information needed to understand the implemented architecture.
- **Information Availability** – assess the availability of information about the architecture

4.3 Properties of SACM

SACM is an extension of CML, which has a structure corresponding to the setup of the general FCM-model. Furthermore, SACM has been validated in Lilienthal's dissertation and has new additional criteria and factors based on cognitive theory from Hutchins[15] and system attributes of Bouwers *et al.*[6]. Hence, SACM is a formal model based on both theory and practice. Bouwers *et al.* also made a claim about the potential deployment of SACM as a model inside Software Risk Assessment process of the Software Improvement Group[7]. This Software Risk Assessment takes two facts into account. The primary facts are obtained through automatically analyzing the source code of a system and secondary facts from people working with or on the system, and available documentation[24]. Most of those facts can be retrieved using SACM. Only the secondary facts have to be retrieved manually by interviewing the stakeholders[24]. Since the full potential of SACM can be accessed from both designed and implemented architecture, SACM can also be deployed as "formal backup for the Light-weight Sanity Check for Implemented Architectures"[5], which has been already suggested in [7]. However, only the factors "Modularity" and "Ordering" of SACM are accessible in case of not existing implemented architecture. We can not exploit the fully potential of SACM at the early stage of development. In oder to provide a balanced assessment of SACM criteria, there is a need of new metrics together with theirs thresholds depending on the threshold approaches[7].

Beside the positive properties, there are several limitations of SACM[6]. One limitation is that designed and implemented architecture are necessary for the evaluation and especially the implemented architecture. Otherwise, the factor "pattern conformity" won't be fully accessed. Another limitation is the lack of a formal proof on the completeness of SACM. There might be factors and criteria, which have not captured in SACM yet. However, the "understanding" in cognitive science, in which SACM is based on, hasn't been proven on its completeness neither[6]. Hence, it is not a real limitation[7].

5. CONCLUSIONS

In this paper, we introduced the definition of software complexity by IEEE and two definitions of architectural complexity introduced by Wang and Lilienthal. Since the architectural complexity is a part of the overall software complexity and its definition can be implied by the definition of software complexity, that might be the reason, why the definition of architectural complexity has not established yet. However, the definitions introduced by IEEE[1] and Wang[25] does not cover all the criteria of the architectural complexity introduced in Lilienthal's dissertation. Thus, those definitions do not define architectural complexity properly. Lilienthal'definition is, however, based on an a more general approach of software complexity comparing to the definition of IEEE and Wang.

The architecture community agrees with relation between cohesion, coupling and architectural complexity. The classification by Stevanetic *et al.* and the formal model of architectural complexity SACM reflect that relation. Besides coupling and cohesion, there are other factors determining the architectural complexity presented in SACM. Since the implemented architecture is required in order to access those factors, only cohesion and coupling are available for assessing architectural complexity in the early development phase, which has been already suggested in [11]. All in all, coupling and cohesion are attributes of architectural complexity, which can be measured during the entire development process.

As the first assessment of structure-based metrics, Stevanetic *et al.* focused on metrics measuring architectural understanding from the deployment view. Most of the metrics are applicable in real projects. Stevanetic *et al.* identified the need for further verification and improvement of those metrics. Although the Stevanetic *et al.*'s work is limited to the metrics measuring from the deployment view, their classification might be the first step towards defining the taxonomy of structure-based metrics measuring understandability.

During the research, we found two approaches on modeling architectural complexity. One approach is the adaption of Wood's task complexity model for the software domain introduced by Darcy *et al.*. Another approach is the Software Architecture Complexity Model(SACM) introduced by Bouwers *et al.*. SACM is a formal model based in both theory and practice. All in all, the structure-based metrics and SACM have a good potential for deployment in a real project. Nevertheless, there is a need of research on new metrics as well as the validation of current structured-based metrics.

6. REFERENCES

[1] *IEEE Standard Glossary of Software Engineering Terminology.* 2011.

[2] E. B. Allen. Measuring graph abstractions of software: An information-theory approach. *Eighth IEEE Symposium on Software Metrics*, June 2002.

[3] E. B. Allen, S. Gottipati, and R. Govindarajan. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. *Software Quality Control*, 15(2):179–212, March 2007.

[4] L. Bass, P. C. Clements, and R. Kazman. *Software Architecture in Practice, Third Edition.* Addison-Wesley Professional, 2012.

[5] E. Bouwers and A. v. Deursen. A lightweight sanity check for implemented architectures. *IEEE Software*, 27(4):44–50, 2010.

[6] E. Bouwers, J. Visser, and A. v. Deursen. Criteria for the evaluation of implemented architectures. *25th International Conference on Software Maintenance*, pages 73–83, 2009.

[7] E. Bouwers, J. Visser, C. Lilienthal, and A. v. Deursen. A cognitive model for software architecture complexity. *18th IEEE International Conference on Program Comprehension*, July 2010.

[8] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies.* Addison-Wesley, 2005.

[9] D. P. Darcy, C. F. Kemerer, S. A. Slaughter, and J. E. Tomayko. The structural complexity of software: An experimental test. *IEEE Transactions on Software Engineering*, 31(11):982–994, November 2005.

[10] C. Ebert. Complexity traces - an instrument for software project management. *International Thomson Computer Press*, pages 166–176, 1995.

[11] M. Galster, A. Eberlein, and M. Moussavi. Early assessment of software architecture qualities. *Second International Conference on Research Challenges in Information Science*, pages 81–86, 2008.

[12] A. Ghazarian. A theory of software complexity. *4th SEMAT Workshop on a General Theory of Software Engineering*, 2015.

[13] V. Gupta and J. K. Chhabra. Package coupling measurement in object-oriented software. *Journal of Computer Science and Technology*, 2009.

[14] V. Gupta and J. K. Chhabra. Package level cohesion measurement in object-oriented software. *Journal of the Brazilian Computer Society*, 2012.

[15] E. Hutchins. *Cognition in the wild*. MIT press, 1996.

[16] R. Karzman and M. Burth. Assessing architectural complexity. *2nd Euromicro Working Conference on Software Maintenance And Reengineering*, 1998.

[17] J. Lankford. Measuring system and software architecture complexity. *in Proc. 2003 IEEE Aerospace Conference*, March 2003.

[18] C. Lilienthal. *Komplexität von Softwarearchitekturen – Stile und Strategien*. PhD thesis, University Hamburg, 2008.

[19] P. K. McCall, J. A. abd Richards and G. F. Walters. Factors in software quality. *US Rome Air Development Center*, 1977.

[20] M. Nagl. *Softwaretechnik: Methodisches Programmieren im Großen*. Springer, 1990.

[21] R. S. Sangwan, L.-P. Lin, and C. J. Neill. Structural complexity in architecture-centric software evolution. *Computer*, 41(10):96–99, October 2008.

[22] S. Stevanetic and U. Zdun. Exploring the relationships between the understandability of architectural components and graph-based component level metrics. *14th International Conference on Quality Software*, 2010.

[23] S. Stevanetic and U. Zdun. Software metrics for measuring the understandability of architectural structures – a systematic mapping study. *19th International Conference on Evaluation and Assessment in Software Engineering*, April 2015.

[24] A. van Deursen and T. Kuipers. Source-based software risk assessment. *ICSM 2003: Proceedings of the International Conference on Software Maintenance*, 2003.

[25] Y. Wang. Cognitive complexity of software and its measurement. *5th IEEE Int. Conf. on Cognitive Informatics*, 2006.

[26] R. Wood. Task complexity: Definition of the construct. *Organizational Behavior and Human Decision Processes*, 37:60–82, 1986.

A Survey on Multi-objective Regression Test Optimization

Karl Ricken
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
karl.ricken@rwth-aachen.de

Andrej Dyck
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
andrej.dyck@swc.rwth-aachen.de

ABSTRACT

Testing is a crucial part of software engineering in which regression testing verifies that functionality did not break with changes. Automating the execution of those checks reduces manual workload. However, in large software applications, the execution of all test cases can still be time consuming and is often unnecessary considering that a change affects only a part of the application. Thus, researchers and practitioners proposed approaches that optimize the execution of automated regression tests; primarily, the reduction of the execution time. It is only in recent years that approaches consider more than one objective in their optimization. These approaches are called multi-objective regression test optimization (MORTO).

MORTO approaches seem promising. Not all of them, however, are clearly labeled as MORTO or even as regression test optimization. This paper reviews recent publications and presents a selection of state-of-the-art MORTO approaches. Furthermore, it discusses two promising techniques and gives a perspective for future research.

Keywords

genetic algorithm, linear programming, morto, multi objective regression testing, regression test, regression test optimization, search-based testing, test prioritization, test selection, testing

1. INTRODUCTION

While developing large software systems, software testing is one of the very important tasks to ensure quality. Regression testing verifies the functionality after code changes. It can be automated with the help of testing frameworks, for example, JUnit for the software language Java.

With an increasing number of automated test cases for a software, the execution of these test cases takes more time and resources. Regression test optimization (RTO) techniques optimize the execution to reduce time and resources of regression testing.

Two groups of techniques, regression test selection (RTS) and regression test prioritization (RTP), currently have high academic relevance for RTO [25]. RTS techniques pick test cases that are relevant for the change and ensure full fault detection. RTP techniques sort test cases in order to detect faults as fast as possible [25].

Many RTO techniques pursue a single objective. Often, RTS techniques minimize the execution time, while RTP techniques order test cases according to coverage or critical-first [25].

In recent years, new techniques considered more than one objective for a better optimization, e.g., reducing costs for database access and maximizing code coverage [8]. RTO techniques that optimize for two or more objectives are called *multi-objective regression test optimization* (MORTO) [8]. Multi-cost optimization and multi-value optimization are also part of MORTO [8].

Techniques that optimize multiple objectives are not always labeled as MORTO. For example, Maia [16] describes a technique that meets the definition of MORTO, but named it RTS and does not mention the term of MORTO. Therefore, this paper surveys the state-of-the-art of MORTO approaches and presents the most promising techniques. Two of these techniques will then be discussed with respect to their relevance for RTO.

The paper is divided into seven sections: After this introduction section 2 describes the methodology used for this survey. Section 3 presents first RTO techniques that consider more than one objective. Section 4 shows the state-of-the-art of RTO techniques that can be labeled as MORTO and the most promising approaches that optimize multiple objectives. In section 5, this paper discusses two promising MORTO techniques and emphasizes why two objectives that have rarely been used until now need more attention. Section 6 describes the threats to validity and section 7 ends this paper with a conclusion.

2. METHODOLOGY

This section presents the tools, keywords and approaches that were used to create this survey. We limited our research to academic papers published between 2009 and 2016 in order to provide a literature review on latest MORTO techniques. Other material like books or magazines was not taken into account.

We started to search for literature in several online research tools and databases, including Google Scholar, CiteSeerX, IEEEXplore, and Mendeley. Search terms included the keywords *genetic algorithm*, *linear programming*, *morto*, *multi objective regression testing*, *regression test*, *regression test optimization*, *search-based testing*, *test prioritization*, *test selection*, and *testing*. To narrow down the search results, we also combined keywords in various ways, for example:

- *genetic algorithm regression test optimization*,
- *search-based testing regression optimization*,
- *linear programming regression test optimization*,
- *multi objective regression testing.*

Out of several million search results, we identified 19 academic papers as relevant for the topic at hand.

In a next research round, we used a snowball technique and identified further relevant papers by analyzing the references of the 19 initial papers. In addition, we used the Google Scholar function *cited by* (recursively) to find further academic papers that cited the initial papers.

3. BASIC TECHNIQUES

For a better understanding of the concepts in the following sections, this paper presents a brief explanation of basic objectives and techniques in RTO.

Harman [8] subdivides types of objectives into two categories: value- and cost-based objectives.

In RTO, cost-based objectives are aimed to be minimized [8]. Examples for RTO-associated costs include, but are not limited to setup, execution, and simulation. Setup costs appear for instance when a test case requires devices, files, or services, before the actual test can be executed. Other costs may appear because developing a simulation of a real system takes a lot of time and money [8].

On the contrary, value-based objectives, also known as benefits, are aimed to be maximized in RTO [8]. Selected examples for RTO-associated benefits are code-based coverage (e.g. branch coverage, mutation coverage, statement coverage) and business sensitivity. Both of these are related to target-market acquisition or revenue generation. Another value-based objective is fault model sensitivity. By its nature, a reasonable RTO-technique should be sensitive to fault models. If it is known that certain faults are likely to occur, then these should be incorporated into the RTO. The tests can then reveal categories of faults that are more likely to be selected or prioritized [8].

Maximizing values/benefits and minimizing costs can be considered as objectives in itself. Because of the limited scope of this paper, however, the authors focus on value- and cost-based objectives.

In a first empirical study on MORTO by Elbaum et al. [3], execution time and cost were combined into a single objective [8]; the technique that was used in this study defines and maximizes a fitness function. A fitness function is a type of objective functions that maps an element out of a set to a real number [13]. The fitness function in the first empirical MORTO study is a ratio of $\frac{value}{cost}$[3] under certain constraints. The idea of using a fitness function under certain constraints is known as a linear programming problem (LP-Problem) [2]. In a LP-problem the objectives should be minimized or maximized under certain constraints. Usually, a LP-Problem has more than one solution that is called *pareto optimal* [5, 2]. A solution is pareto optimal if no objective can be optimized without making other objectives worse[5].

Another essential method in software testing is pairwise testing. To avoid testing all possible combinations of parameter assignments, pairwise testing deletes duplicates from each paired combination of a parameter assignment [28].

Another approach to optimize multiple objectives is a weighted objective function

$$WO(x) = w_1 \cdot f_1(x) + \cdots + w_n \cdot f_n(x)$$

which puts n objectives $\{f_1, \ldots, f_n\}$ into a single weighted objective for weights $\{w_1, \ldots, w_n\}$[7]. Note that each function $f_i(x)$ can be a fitness function, too. This technique was used by Wang et al. in 2013 [30]. They defined a function with three objectives: test minimization percentage (measuring the reduction of the number of test cases), feature pairwise coverage (measuring the degree of pairwise coverage that can be achieved by a chosen solution) and fault detection capability (measuring the fault detection capability of a selected test solution for a product)[30]. Their fitness function is based on $w_1 = \frac{1}{3}, w_2 = \frac{1}{3}, w_3 = \frac{1}{3}$. These values were defined in cooperation with test engineers of Cisco. Wang et al. extended this technique for Cisco in 2014. They defined a fitness function for a single cost objective (for example overall execution cost) and several effectiveness measures (Wang et al. [31] used prioritized extent within a given test resource, feature pairwise coverage, and fault detection capability). Also, each objective got the same weight.

Determining the optimal weights for the objectives, however, is not always possible. A pareto optimal approach is generally better than the weighted objective function because it manages the relation between value- and cost-based objectives [6].

4. CONCRETE TECHNIQUES

Most of the techniques that can be labeled as MORTO are RTS and RTP techniques that pursue multiple objectives. This section presents promising techniques in RTS and RTP, as well as hybrid approaches that are a combination of both, RTS and RTP. Moreover, it describes further techniques that optimize for multiple objectives and are based on natural observations: Ant Colony Optimization (ACO).

4.1 MORTO in regression test selection

In their empirical study on software testing, Maia et al. [16] minimize the execution time, minimize the risk of test cases, and maximize importance of the requirements covered. For example, the authors declared lack of experience with new tools or technologies as a risk of test cases. They found valid and optimal solutions by the use of a heuristic algorithm that is based on a NSGA-II algorithm [16]. In the future, Maia et al. intend to validate this approach with more tests. Their next step is to validate the technique with a major quantity of data and with other multi-objective meta-heuristics [16].

Another approach is a generic test selection strategy called *THEO* [9]. It ensures that all tests will be executed on all code changes at least once before releasing the software product. The purpose of this technique is to find all code defects. Moreover, it increases the productivity and minimizes the overall testing time [9]. According to the authors, the product quality will not be sacrificed. However, THEO may delay defect detection to later development phases because it skips test cases for which the expected cost of running them exceeds the cost of not running them [9].

At Microsoft, a large industrial software company, many product teams are convinced of the success of THEO [9]. Because of the attention from large industrial companies, it seems likely that this technique will be developed further.

Parsa and Khalilan [20] present another approach:
A greedy algorithm that maximizes the fault detection capability and minimizes the number of test cases. These two objectives are used for test suite minimization, but can also be classified as RTS. First, a test case-requirement matrix will be defined where the elements are either 1 if the test case satisfies the requirement or 0 if not. Through mathematical operations the technique indicates and selects optimal test cases. Optimal test cases satisfy the maximum number of unmarked requirements and have the minimum overlap in requirement coverage. For the future, Parsa and Khalilan intend to conduct experiments on available real C programs as well as java benchmarks [20].

Tyagi and Malhotra [29] present a further approach that prioritizes test cases in three steps. At first, redundant test cases are removed. In the second step, a multi-objective algorithm called *multi objective particle swarm optimization* (MOPSO) selects a minimal set of test cases that pursues the target to cover all faults in minimum execution time. In the last step, the approach calculates the ratio of fault coverage to the execution time of test cases. The higher the ratio, the higher the priority. Finally, the approach orders the test cases by priority [29].

4.2 MORTO in regression test prioritization

Ashraf et al. [1] introduced a new algorithm, called *value-based particle swarm optimization*. The algorithm pursues six objectives: changes in requirement, implementation complexity, requirement traceability, execution time, customer priority, and fault impact of requirement. The authors tested the algorithm and compared the results with the results of a random prioritization. This random prioritization acts as a base-line for comparison [1]. Ashraf et al. tested this approach only on a limited data set. In the future, they intend to validate the swarm optimization by using larger sized projects with a larger number of test cases. Furthermore, the authors intend to investigate their approach with evolutionary techniques in order to show the advantages, for example, the early rate of fault detection [1].

Another approach in RTP that is based on a genetic algorithm was developed by Kaur and Goyal [14]. Genetic algorithm are programs that solve complex problems based on the idea of natural selection [10]. Kaur and Goyal's [14] technique expands RTP by faults covered in minimum execution time and total code coverage. The authors used the Average Percentage of Condition Coverage (APCC) approach to analyze code coverage. In the near future, this algorithm will be implemented in an automation tool for solving large numbers of test cases [14].

Huang et al. [12] used historical records for a
cost-conscious RTP approach. The objectives of this approach are to minimize test costs and to create a strictly formalized approach that can easily be adopted by future work. Huang et al.'s technique orders the test cases by their various test costs and fault severities [12]. Most RTP techniques analyze the source code, but Huang et al. take a different approach. Their technique does not rely on the availability of the source code and can thus be used in projects in which the tester does not have access to the source code. To verify their technique Huang et al. gave an outlook on adding more issues to solve, for example, combining their history-based technique with coverage-based techniques [12].

Marijan et al. [18] developed another RTP approach *Prioritization for Continuous Regression Testing* (ROCKET). With given set of test cases, execution time for each test case, failure status for each test case based on last executions, and total time of testing, ROCKET determines a priority for each test case by using specific heuristics. Those heuristics depend on specific industrial settings [18]. The main objectives are minimizing the execution time and having higher regression fault detection. Furthermore, the authors present an industrial test case study of their approach and demonstrate the resulting effectiveness based on comparison with other techniques. In the future, Marijan et al. intend to extend their approach for more criteria, for example, the cost of switching test cases in execution that require manual intervention or the cost to fix the failure [18].

Marijan [17] developed a further RTP approach that integrates three objectives as criteria for prioritizing the test cases. These objectives include: business perspective, performance perspective, and technical perspective. The author developed his technique by testing industrial mobile device systems that were developed in continuous integration. His approach maximizes the number of executed test cases with high failure frequency, failure impact, and cross-functional coverage [17]. In the future, Marijan intends to extend the approach by considering more prioritization factors. His approach seems promising because it is developed and tested for industrial projects [17].

Strandberg et al. [27] introduced a new approach based on Marijan's [17] technique in cooperation with Westermo Research and Development AB [27]. This technique is called *SuiteBuilder* and orders test cases by the weighted average of the following prioritization methods (called *Prioritizer* [27]):

1. *TestPrioritizer*: experiences from developers are noted in configuration files that are rarely modified.
2. *TagPrioritizer*: each test case is allocated to a test case group. Developers grade those groups.
3. *FailPrioritizer*: if a test case failed in the past, it gets a higher rating than those test cases without a fault.
4. *RecentPrioritizer*: untested test cases are preferred.
5. *SourcePrioritizer*: analysis of detailed logs created by the devices on which the software is running. This analysis can locate code areas that have recent changes [27].

This approach is intended for industry use at Westermo Research and Development AB. It shows the effectiveness of prioritization by considering multiple objectives. Strandberg et al. do not mention plans of expanding their approach, but they expect extension from other companies in the future [27].

Raju and Uma [22] present a further approach that uses a genetic algorithm to weigh seven prioritization factors: customer assigned priority of requirements, implementation complexity, changes in requirements, fault impact of requirements, completeness, traceability and execution time.

The authors evaluate the effectiveness of their technique by using the average-percentage-of-faults-detected (APFD) metric. The motivation of this approach is to reveal more severe faults at an earlier stage and to improve customer-perceived software quality [22].

4.3 Hybrid approaches

In this subsection some approaches that are based on both, RTS and RTP, will be presented. The techniques described before were clearly labeled as RTS or RTP by the corresponding authors. The following techniques are not explicitly labeled as one of the two. They are based on a combination of RTS and RTP techniques or other optimization strategies that are called *hybrid* techniques [23].

Mirarab et al. [19] introduced an approach that is based on different optimization strategies. The authors formalized the problem (optimize multiple objectives) to a LP. The target function is to achieve the given objectives. The constraints are the limit of the test cases that can be tested and the maximal time for testing all test cases. This LP will be solved by approximate algorithmic methods, for example, neighborhood searching. The idea is to first concentrate on one objective; afterwards the technique integrates other objectives step by step [19]. The authors do not mention explicit goals. This technique is helpful for using limited resources more effectively by making small sacrifices to one criterion, enabling noticeable improvements in other criteria [19]. Furthermore, the approach is flexible for new criteria and can be adapted for different objectives. The approach, however, is kept general. That is the reason why Mirab et al. suggest to study it in more depth themselves [19].

Sampath et al. [23] defined a further technique that considers multiple criteria as a hybrid, called *hybrid criteria* [23]. The goal is to apply multi-objective algorithms (e.g. Multi-Objective Genetic Algorithm) that solve the problem: select a pareto optimal subset of the test suite, based on multiple test criteria. Sampath et al. suggest revisiting many techniques that pursue a single objective. In their opinion, many techniques can pursue multiple objectives [23].

4.4 Ant Colony Optimization approaches

Not all techniques are based on well-known algorithms. For example, Ant Colony Optimization (ACO) is a strategy based on observations of ants [11] and is an example for the diverse sources that ideas for RTO (and software testing in general) can come from. The strategy transferred a phenomenon from nature to programming by imitating how ants find the best way to the nearest food source. In nature, all ants spread out to find the closest food source. As soon as one ant finds a source all ants will start to use this source until it is exhausted. Nature thus optimizes for one objective: proximity. Some RTO techniques follow this idea and have been analyzed for industrial projects [1, 15, 24, 26, 29].

ACO is employed in the approach of Singh et al. [24]. The objectives of this approach are the minimization of execution time with minimal effort required and the maximization of fault coverage. Sing et al. also suggest using their technique for larger software projects. The more test cases the closer is the technique to the optimum [24]. The authors plan to implement the technique for automation and apply it to complex and large software. Also, Singh et al. intend to compare their approach with other techniques, for example, genetic based prioritization techniques [24].

Kumar [15] presents an approach to optimize multiple objectives through ACO. The goal of Kumar's technique, however, is to optimize for two objectives. Metaphorically, he modified the system in a way that a few ants will continue to look for a better food source that is not necessarily the closest source. In reality, Kumar's technique enhances the fault-detection rate and minimizes the execution time [15].

Solanki et al. [26] examined the Modified Ant Colony Optimization (m-ACO). M-ACO means that ants select the shortest distance to food but also check the food quality by calculating the food source fitness. The authors implemented and tested such a technique and used it to order test cases in a test suite [26]. This technique is comparable to the technique of Kumar described before.

5. DISCUSSION

In this part, the paper will take a critical look at the existing MORTO techniques. While RTS and RTP approaches have already been examined in more detail in academic papers, ACO and hybrid approaches still lack further academic attention. ACO and hybrid approaches as they were presented in this paper do thus not allow for a substantiated discussion. Moreover, the limitations of this paper do not allow for a full discussion of all techniques that were presented earlier. Because of this, the RTS approach of Parsa and Khalilan [20] and the RTP approach of Strandberg et al. [27] were selected as exemplary techniques for discussion. In addition, setup costs and fault sensitivity will be discussed as exemplary new main objectives in MORTO. These objectives have rather been neglected in the past, but seem promising and should thus get more academic attention in the future.

5.1 Critical review of RTS and RTP approach

Both approaches, by Parsa and Khalilan [20] and by Strandberg et al. [27], seem promising for the future, but currently do not contain criteria that are relevant enough for the future. This subsection provides a critical review of both techniques.

Parsa and Khalilan [20] present a RTS technique that used a greedy algorithm to handle three objectives: maximizing fault detection capability, which also maximizes the number of testing requirements, and minimizing the number of test cases. The authors defined a matrix in which the entries represent test cases and test requirements. In comparably few steps this technique removes test cases that can be neglected if other test cases can cover their requirements [20].

This idea focuses on a minimal set of test cases that satisfies all test requirements, but it does not pay attention to execution time. Indeed, there is a correlation between the number of test cases and the total time for testing. Yet, when the technique has to select one of two test cases that check the same requirements, it does not necessarily select the test case with the shortest execution time. In case of industrial application, however, testers have only a limited time for testing large software. Still, the basic technique is promising because it can be adapted by changing the input objectives of the matrix. The mathematical way for solving problems with matrix operations allows the tester to create two-tuples for all objectives.

Strandberg et al. [27] use historical assessments of test cases. By analyzing historical data the technique can learn and calculate better priorities for each test case [27]. Nevertheless, the tester needs to invest a lot of time to provide detailed information for test cases. Because of that, this technique cannot run independently of human testers.

Both techniques take fault coverage into account and in the end both techniques try to minimize the number of test cases. Furthermore, both techniques are neither able to take other objectives than the referred objectives (maximizing fault detection, minimizing the number of test cases, and minimizing time for cost reduction) into account, nor can they handle more than two objectives at once. A good technique, however, has to be flexible for changing objectives. We think an extension of both techniques could solve this problem: In Strandberg et al.'s [27] technique, for example, we recommend to implement a further Prioritizer[27] (see Section 4.2). This new Prioritizer could consider the priority of a new objective such as setup costs. Moreover, an extended version of Parsa and Khalilan's [20] technique could solve two-tuples for all objectives. Each tuple contains a test case and an objective. Mathematical operations could then combine the newly created matrices.

5.2 Significance of two neglected objectives

Because of the relevance and effectiveness of RTO [32] and the diversity of RTO tasks in big projects, more and better approaches are needed. Currently, time reduction and code coverage are the two main objectives in MORTO. Two other objectives, however, should also be given more research attention: setup costs and fault model sensitivity.

Services (e.g. a web service, where two electronic devices communicate with each other), files and devices are needed just as much for test cases as user inputs [8]. Test cases for web applications often use interaction with files from the user. Therefore, it is essential to take care of services, files and devices, for example, upload and download files. There is no single technique that considers these setup costs. Moreover, it is also necessary to take fault model sensitivity into account. Many faults are similar. It is useful to select test cases that reveal categories that are more likely to include faults. Place test cases at the beginning where costs are zero. For example, test cases that test an empty database.

The combination of minimizing setup costs, maximizing fault model sensitivity, and ordering the test cases with a high risk of faults at the beginning bears another benefit: Most known faults can be detected at the beginning of the regression test. This leads to lower testing time. Also, setup costs are affected: Finding faults at the beginning saves the setup costs of useless and dependent test cases.

6. THREATS TO VALIDITY

In this section, we describe the potential threats to validity of the survey. As outlined in Section 2 we used only the addressed online tools. There are also many other sources, tools and material (books, magazines, etc.) that might contain more techniques and relevant information.

Furthermore, the selection of academic papers that this survey is based on represents another threat to validity. Because of the high number of search results in the initial search round, only a certain number of those papers could be analyzed in detail. For example, the search term *testing* gives 5.130.000 results. Hence, we used combinations of the keywords that were described in Section 2. For every keyword and/or combination, we limited ourselves to quick-read only about 200 papers. Out of these 200 papers we then selected the 1 to 3 most relevant papers for every keyword. By skimming through the papers, techniques that are not clearly labeled as MORTO could have been missed. To limit the effect of this threat, we used several keywords and combinations of keywords that are related to and commonly used in context with MORTO. However, compared to the high number of search results, only a very small number of academic papers was finally used as a basis for this survey. It is likely that there is further relevant information on MORTO, e.g., further MORTO techniques, that have not been included in this survey.

Moreover, not all techniques might be generalized and used for e.g., industrial problems. The MORTO techniques presented in this survey stem from academic settings and cannot unconditionally be transferred to e.g., industrial settings. This threat might be reduced if industrial practitioners adapt the presented techniques for their specific objectives and test larger software with more test cases.

7. CONCLUSIONS

At the moment, some companies already use MORTO techniques, for example, Cisco Systems [21] and Microsoft [9]. Because of the high industrial relevance of the topic, the number of academic articles published on MORTO is soaring. This shows that the importance and acceptance of MORTO is increasing, especially because it combines established techniques like RTS and RTP. So far, however, the MORTO techniques are mainly used in bigger software projects. Future research could concentrate on techniques that could help software developers with verifying also smaller software projects.

This paper presents a selection of approaches that can be classified as MORTO. As the discussion shows, more and better approaches are useful to optimize other kinds of software projects than those that MORTO is currently used for. MORTO is not only an idea of RTO techniques, but also a crucial part for minimizing the cost and maximizing the benefit of RTO at the same time. Because of the observance of multiple and different objectives, MORTO is likely to become an integral part in software testing.

There are also other techniques in software testing that are not part of RTO. Gueorguiev et al., for example, introduced a search based approach to software project robustness [4]. Their goals were to minimize the completion time, which ensures the early time to market, and to build software robustness simultaneously. This shows that the relevance of the idea of MORTO (optimize regression testing for two or more objectives) [8] is used in many parts of software testing.

8. REFERENCES

[1] E. Ashraf, a. Rauf, and K. Mahmood. Value based Regression Test Case Prioritization. *Proceedings of the World Congress on Engineering and Computer Science*, I, 2012.

[2] H. Baller, S. Lity, M. Lochau, and I. Schaefer. Multi-objective test suite optimization for incremental product family testing. *Proceedings - IEEE 7th International Conference on Software Testing, Verification and Validation, ICST 2014*, pages 303–312, 2014.

[3] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 329–338, 2001.

[4] S. Gueorguiev, M. Harman, and G. Antoniol. Software Project Planning For Robustness And Completion Time In the Prescence Of Uncertainty Using Multi Objective Search Based Software Engineering. *11th Annual Conference on Genetic and Evolutionary Computation, GECCO 2009.*, pages 1673–1680, 2009.

[5] I. Halevy, Z. Kava, and T. Seeman. Normalization and Other Topics in Multi Objective Optimization. *Proceedings of the Fields-MITACS Industrial Problems Work shop*, 2:89–101, 2006.

[6] M. Harman. The current state and future of search based software engineering. *FoSE 2007: Future of Software Engineering*, pages 342–357, 2007.

[7] M. Harman. Why Source Code Analysis and Manipulation Will Always Be Important. *10th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 7–19, 2010.

[8] M. Harman. Making the case for MORTO: Multi objective regression test optimization. In *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, pages 111–114, 2011.

[9] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. *Proceedings - International Conference on Software Engineering*, 1:483–493, 2015.

[10] J. H. Holland. Genetic Algorithms - Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand, 1992.

[11] A. E. Howe, A. von Mayrhauser, and R. T. Mraz. Test Case Generation as an AI Planning Problem. *Automated Software Engineering*, 4:77–106, 1997.

[12] Y. C. Huang, K. L. Peng, and C. Y. Huang. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, 85(3):626–637, 2012.

[13] T. Jansen. *Analyzing Evolutionary Algorithms: The Computer Science Perspectiv.* Springer Science & Business Media, 2013.

[14] A. Kaur and S. Goyal. a Genetic Algorithm for Regression Test Case Prioritization Using Code. *International Journal of Advanced Trends in Computer Science and Engineering*, 3(5):1839–1847, 2011.

[15] S. Kumar. Modified ACO to maintain diversity in Regression Test Optimization. *3rd International Conference on Recent Advances in Information Technology (RAIT)*, 2016.

[16] C. L. B. Maia, R. A. F. Do Carmo, F. G. De Freitas, G. A. L. De Campos, and J. T. De Souza. A Multi-Objective Approach For The Regression Test Case Selection Problem. *XLI Brazilian Symposium of Operational Research, XLI SBPO 2009.*, pages 1824–1835, 2009.

[17] D. Marijan. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. *Proceedings - 2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015*, pages 157–162, 2015.

[18] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. *IEEE International Conference on Software Maintenance, ICSM*, pages 540–543, 2013.

[19] S. Mirarab, S. Akhlaghi, and L. Tahvildari. Size-constrained regression test case selection using multicriteria optimization. *IEEE Transactions on Software Engineering*, 38(4):936–956, 2012.

[20] S. Parsa and A. Khalilian. On the Optimization Approach towards Test Suite Minimization. *International Journal of Software Engineering and Its Applications*, 4(1):15–28, 2010.

[21] D. Pradhan. Test Optimization using Weight Based Search Algorithms in a Maritime Application. *Institutt for informatikk, University of Oslo*, 2015.

[22] S. b. Raju and G. Uma. Factors oriented test case prioritization technique in regression testing using genetic algorithm. *European Journal of Scientific Research*, 74(3):389–402, 2012.

[23] S. Sampath, R. Bryce, and A. M. Memon. A uniform representation of hybrid criteria for regression testing. *IEEE Transactions on Software Engineering*, 39(10):1326–1344, 2013.

[24] Y. Singh, A. Kaur, and B. Suri. Test case prioritization using ant colony optimization. *ACM SIGSOFT Software Engineering Notes*, 35(4):1, 2010.

[25] Y. Singh, A. Kaur, B. Suri, and S. Singhal. Systematic literature review on regression test prioritization techniques. *Informatica (Slovenia)*, 36(4):379–408, 2012.

[26] K. Solanki, Y. V. Singh, and S. Dalal. Experimental analysis of m-ACO technique for regression testing. *Indian Journal of Science and Technology*, 9(30), 2016.

[27] P. E. Strandberg, D. Sundmark, W. Afzal, T. Ostrand, and E. Weyuker. Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors. *27th International Symposium on Software Reliability Engineering*, 2016.

[28] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109–111, 2002.

[29] M. Tyagi and S. Malhotra. Test case prioritization using multi objective particle swarm optimizer. *2014 International Conference on Signal Propagation and Computer Technology, ICSPCT 2014*, pages 390–395, 2014.

[30] S. Wang, S. Ali, and A. Gotlieb. Minimizing test suites in software product lines using weight-based genetic algorithms. *2013 15th Genetic and Evolutionary Computation Conference, GECCO 2013*, pages 1493–1500, 2013.

[31] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen. Multi-objective test prioritization in software product line testing. *Proceedings of the 18th International Software Product Line Conference on - SPLC '14*, pages 32–41, 2014.

[32] S. Yoo, R. Nilsson, and M. Harman. Faster Fault Finding at Google Using Multi Objective Regression Test Optimisation. *Fse*, 2011.

Current State of Testing Infrastructure as Code

Jens Böttcher
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
jens.boettcher@rwth-aachen.de

Andreas Steffens
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
andreas.steffens@swc.rwth-aachen.de

ABSTRACT

Infrastructure as Code (IaC) is a widely-used approach for defining software environments and infrastructure, but testing IaC is not well researched yet. In this paper, we show current tools and proposed approaches for testing IaC. We do this by first giving a definition of Infrastructure as Code and also a short overview of software engineering testing techniques, which could be applied to IaC. Thereafter, current approaches for testing Infrastructure as Code are discussed. Then, we show promising proposed approaches on testing IaC. In the end we conclude that software engineering testing techniques can be applied to IaC, but we identified gaps and propose ideas how to fill these.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.9 [**Software Engineering**]: Management—*Software configuration management*

Keywords

Infrastructure as Code, configuration management, testing, test generation

1. INTRODUCTION

"[When utilizing Infrastructure as Code,] we can start to think about our infrastructure as redeployable from a code base, in which we are using the same kinds of software development methodologies that have developed over the last 20 years as the business of writing and delivering software has matured." [13]

Nowadays, many software products relay on remote services like databases and file servers. These services are also part of the software products and are called the infrastructure of the project; without these services, the product itself can not function correctly. Hence, the infrastructure of a software product is an important part of the project as well. Developers and system adminstrators tend to manage infrastructure manually, but this can lead to an unknown state of the infrastructure, sudden outages or a *snowflake server* (i.e. servers, which no ones dares to touch because of the fear to break something). [12] *Infrastructure as Code* (IaC) is a practice from DevOps to define software infrastructures as code to make them reliable, deterministic and reproducible. [6] During the life of a project, the infrastructure of the application may change (e.g. switching database versions or servers).

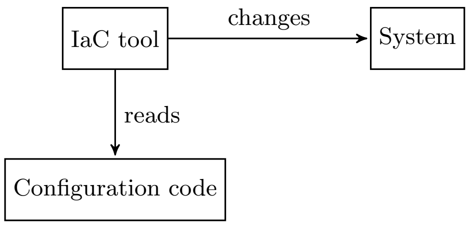


Figure 1: Overview of interactions between an IaC tool, the configuration and the system.

With IaC, the configuration of the infrastructure of the software product is defined in human-readable code. This code can be managed via version control systems to keep it consistent but also to make it accessible to everyone working on the product. But not only does IaC keep the infrastructure consistent, it also helps reducing costs (due to less downtime), speeds up development (due to clear definitions of the infrastructure to develop for) and lower risk of errors and security violations. [19]

Figure 1 shows an overview of the interactions between the components of IaC. The IaC tool reads to configuration code and changes the system in order to fit it to the configuration.

IaC tools evolved from scripts written by system admins to manage their own systems to now full featured frameworks. These tools provide methods for managing a system, like installing packages and configuration changing of the system and applications. They run on different OS but provide the same methods for every OS they run on. For example, using a method `install` to install nginx will install this application on the system, independent of how this application is named in the OS's package manager. IaC tools use either common data notation languages (e.g. JSON, YAML), common programming languages or their own domain-specific language for configuration definition. Popular IaC tools include *Chef* [4], *Puppet* [14] and *Ansible* [2], which we will focus on in this paper. Chef configuration is written in Ruby, Puppet uses a custom DSL which is based on Ruby and Ansible configuration is written in YAML.

Infrastructure as Code also includes orchestration of multiple systems simultaneously. For example, controlling a container cluster is possible with tools like *Kubernetes* [9] or *Docker Swarm* [17]. In this paper, we will focus on IaC tools for single system provisioning. We argue that most of the discussed testing techniques can also be applied to other IaC fields.

Motivation

One of the most important fields in software development is testing. [11] Software can be tested in different complexity (e.g. units, whole application) and various aspects (e.g. security, features). In his master thesis "*Evaluating the testing quality of software defined infrastructures*" [16], Ben Siebert analysed which software quality aspects should also be applied to IaC. He did this by trying to apply a quality model for traditional software to IaC software. Also, he interviewed practitioners of IaC to get their insight on quality models of IaC. Siebert came to the conclusion, that testing is most important quality to seek in IaC software, even more important than automation and best practices.

There already exists a wide range of testing tools for IaC, some of which will we present briefly in this paper. Later we show current research on automated generation of test cases for IaC. Therefore, we aim to analyse the current state of testing for *Infrastructure as Code* in this paper.

In the next section, we will give some general definitions needed for the rest of the paper as well as some definitions on software testing in section 2.1. In section 3 we will present current and proposed approaches on testing IaC. Section 3.1 will focus on frameworks which are already widely used, section 3.2 will present proposed approaches for generating test cases from configurations. Then, in section 4 we will discuss testing techniques which are currently not covered. In section 5, we will conclude this paper.

2. DEFINITIONS

Before proceeding further, we need to make some definitions which are needed for understanding the remainder of this paper.

A system consists of many parts: installed packages, present users and files. A *resource* defines what state a specific part of the system should reach. By applying a resource on a system, the state of this part of the system changes such that the resource is satisfied. Each resource is declarative: it only defines what state should be reached but not how to get there. How the state is reached depends on the underlying operation system and is done by the IaC tool. For instance, a resource named `file` might be used to create, change or delete a file within the file system.

Listing 1 is an example for a `file` resource written in YAML for Ansible. Directory `/opt/some_directory` will be created on the system if it is not already present. It is also possible to define dependencies between resources (e.g. resource `unzip` might depend on resource `download` to download an archive before unzipping it). A *configuration* is a set of resources to be applied on a system. Such a configuration

```
# Create a directory if doesn't exist
- name: Create directory
  file:
    path: /opt/some_directory
    state: directory
```

Listing 1: Example of Ansible's file resource, which will create a directory of not present.

is executed by the corresponding IaC tool.

When applying a resource on it's desired state and returns the same state (i.e. the state is unchanged), the resource is said to be *idempotent*. In other words, applying an idempotent resource successively on a state will change the state at most once.

Furthermore, when applying a set of resources, in any order, successively on a system and the system's final state after each run changes at most once, the system *converged* into this state. It follows that a convergent configuration is also idempotent. [5]

State transition graphs (STGs) are special graphs where the nodes represent states and the edges are directed and represent actions to transition from one state to another. STGs can be used to represent the state of a machine (nodes) and the applied resources of a configuration (edges). In this paper, only connected acyclic graphs are considered.

2.1 Software testing

Testing software is an important and big field in software engineering. Tests can be divided in the complexity of the tested code (i.e. from testing only small units like functions up to testing the whole application), which aspects are covered (e.g. verification of met requirements, met security standards) and either testing static or dynamic (source code versus running/compiled application). [11]

Complexity of code which can be tested, defined by the Software Engineering Body of Knowledge [1]:

Unit testing Small units; single functions or classes are tested

Module testing Coupled elements which form a module, or package, are tested

Integration testing Tests if components of project cooperate correctly

System testing Test if system meets its requirements

Apart from the complexity of a test, tests can also be differentiated by software engineering aspects they cover. For keeping it short, we are only listing those we think are applicable to IaC:

Functional testing.

Verify that all specifications of the software are met. [11] For IaC, this would mean to test if the configuration holds the specifications previously defined.

Conformance testing.

Test if compliance, security and policy requirements are met by the system. [11] For IaC, meeting security requirements is critical because most target systems of IaC are system accessible over the internet.

Destructive testing.
Causing the system to fail with destructive actions (e.g. disable network access, delete files). [3] These kind of tests would verify that the target system would still reach it's desired state even if something went wrong during convergence.

Idempotence testing.
A function should return the same result for the same input on every sequential execution. [7] This is important for IaC because it would verify that a system, which is already in it's desired state, does not change.

State-based testing.
Models behaviour of the system as state machine and simulate system within this machine. [18] Because resources describe system states, such testing is suitable for IaC.

A/B testing.
A/B testing [8] could be used for testing different input systems. While mostly used as tool for optimizing marketing with such testing, it could be determined if a configuration is dependent on a specific input system state.

Compatibility testing.
Similiar to A/B, compatibility testing [20] would test if the configuration works on multiple different operation systems.

It can also be differentiated between dynamic and static testing: running tests against a compiled or running application is dynamic testing, whereas static testing is testing of the source code (e.g. linting). [10]

In the context of Infrastructure as Code, dynamic testing can also be called post-convergence testing because a configured system is tested. Consequently, static testing can be called pre-convergence. In the following, we will use the terms pre-convergence and post-convergence.

3. TESTING INFRASTRUCTURE AS CODE

"Systems should be in place to ensure that one's code produces the environment needed and that any changes have not caused side effects that alter other aspects of the infrastructure." [13]

In this section we discuss different approaches for testing IaC. As stated in section 2.1, software testing is a critical task in software development. Usually, the project's application is tested but the project's infrastructure code is not, which can lead to problems (e.g. sudden outages due to missing stress tests). Also because IaC reduces costs, speeds up development and lowers risk of errors and security violations, keeping the infrastructure and its code tested is essential. [19]

In the remainder of this section, we will first give an overview of currently used testing framework for IaC. Then, in section 3.2, we discuss theoretical approaches, which are still in the prototype phase.

3.1 Current frameworks

Because pre-convergence testing analysis the code and each tools uses a different language to define configurations, tools for pre-convergence need to be suited for the specific IaC tool at hand. Hence, for each of the popular IaC tools there are multiple pre-convergence testing tools. We will list some for each of the three IaC tools.

Then we give examples for post-convergence tools which are suitable for all IaC tools, sometimes with few modifications.

Pre-Convergence

Here we list tools for making pre-convergence testing on configuration, divided by target IaC tool. These tools employ conformance testing since they check the code against external policy requirements (linting rules) for the code.

Chef.
RuboCop[1] is a linting tool for Ruby, which help preventing well-known pitfalls during programming based on a community-driven Ruby style guide.[2]

Foodcritic[3] is a linting tool specifically written for Chef, using their own set of rules for preventing common problems.

Puppet.
Linting for Puppet configurations can be done with Puppet-lint[4], which tests against the Puppet's official style guide.[5]

Ansible.
yamllint[6] can be used to lint the YAML files syntax error as well as Ansible-lint[7] for testing against Ansible's best practice guide.[8]

Post-Convergence

Because post-convergence describes the stage in which the configuration is applied to the system, testing in this stage can be done independently on the used IaC tool.

Serverspec[9] is a Ruby framework for running tests against running system. Hence, it can be used for post-convergence testing, either on a real system or a test system. It enables functional testing.

Test Kitchen[10] utilizes containerization and cloud providers for running configurations against test systems running specified OS's. Despite it being aimed at Chef configurations, there also exist plugins to enable use with Puppet and An-

[1] `https://github.com/bbatsov/rubocop`
[2] `https://github.com/bbatsov/ruby-style-guide`
[3] `https://github.com/acrmp/foodcritic`
[4] `https://github.com/rodjek/puppet-lint`
[5] `https://docs.puppet.com/puppet/latest/style_guide.html`
[6] `https://github.com/adrienverge/yamllint`
[7] `https://github.com/willthames/ansible-lint`
[8] `https://docs.ansible.com/ansible/playbooks_best_practices.html`
[9] `http://serverspec.org`
[10] `http://kitchen.ci`

sible. Like Serverspec, Test Kitchen can be used to execute functional testing.

InSpec[11] was created as a plugin for Serverspec but was later developed into a standalone application. In contrast to Serverspec, InSpec is more aimed at conformance testing.

To our knowledge, there are currently no tool for testing the idempotence of a configuration, which is covered by the approaches presented in the next subsection.

3.2 Theoretical frameworks

In this section, we discuss two approaches for testing IaC which are currently in prototype phase: section 3.2.1 presents a post-convergence approach, whereas Section 3.2.2 focuses on a pre-convergence approach. Both of these approaches focus on generation of test cases and testing the idempotence of the configuration.

3.2.1 Post-Convergence

Hummer et al. [5] proposed a technique for dynamically testing idempotence and convergence of partially ordered configurations, using state transition graphs (STGs).

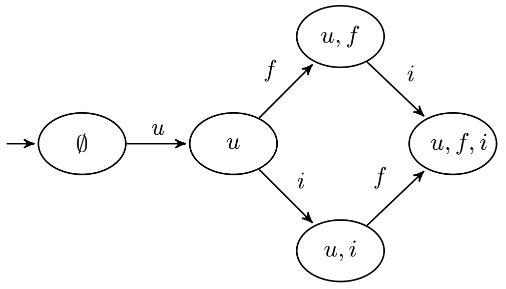
Partially ordered configurations are configurations with partially defined order, where the IaC uses an arbitrary order to execute, in regard to the defined dependencies across the resources. Their approach converts a given configuration into an STG and then leverage from the properties of an STG to generate test cases.

From the dependencies across the resources of the configuration, for each resource r sets of *ancestors*, *successors* and *non-related* resources are defined. Ancestors are all resources r depends on, successors are all resources which depend on r and non-related resources are the remaining resources. Possible execution orders are defined from ancestors and successors of each resources. All ancestors of r must be executed before r, and r before its successors. The authors state that "the problem space of possible execution orders can be huge" but argue that "there are efficient polynomial time approximation algorithms" for this problem.

Resource b *preserves* resource a iff applying b after a will still hold the desired state of a (i.e. applying a again after b will not change the state). It follows, that a configuration with idempotent resources and preservation of their respective ancestor and non-related resources, is convergent. Also, any successful execution of such a configuration will reach a state which will satisfy every resource of the configuration.

Using the possible execution orders of a configuration, an partitioned STG is generated. In this partitioned STG, each resource r is represented by an edge and every node is the set of resources from the previous node joined with r. The initial state of the system is denoted as an empty set, which is assumed to be a freshly setup test system.

Let's say we have three resources: user creation (u), creation of files in the new user's home directory (f) and installation of an application package (i) which will later be run by the new user; all three are idempotent, preserve each other and f and i depend on u. Figure 2 shows the resulting partitioned STG. Note that the system will eventually reach state u, i because of the relation u and i have to another. Every possible path in the STG represents a test case. Multiple approaches for path finding are possible, listed are path



$$\begin{tikzcd} & & u,f \arrow[dr, "i"] & \\ \arrow[r] & \emptyset \arrow[r, "u"] & u \arrow[u, "f"] \arrow[d, "i"] & u,f,i \\ & & u,i \arrow[ur, "f"] & \end{tikzcd}$$

Figure 2: Partitioned STG of a configuration with three resources.

coverage and edge coverage.

A prototype implementation of this approach is available on GitHub[12], which is a Ruby application primarily focused on testing Puppet configurations. The authors state, that it would be possible to modify their implementation in order to work with Chef or Puppet.

For verification of correctness of the current state in regards to the executed resource, the file system is watched and after execution of each resource the changes of to the file system are compared to the expected changes, generated by the testing framework. It is, however, unclear how these expected changes are generated.

For evaluation, the authors have selected real world 101 Puppet configurations and additionally 11 configurations with known bugs, which then were automatically tested in isolation inside Docker containers. Overall, 250.805 test cases were generated with a net execution time of 9.15 days. Test results of currently 151 configurations are available.[13]

Most problems arose when resource were depending on another: "If two or more resources manage the same aspect of the system, conflicts occur if the fail to coordinate themselves and do not agree on a shared desired state." Also, resources depending on other resources without the declaration of this dependency yield different outcomes if the execution order is changed.

3.2.2 Pre-Convergence

Shambaugh et al. [15] presented a static verification tool, named Rehearsal, with test case generation for Puppet code.

Similar to [5], the authors generate a directed acyclic graph from the given configuration. In contrast, here resources are represented by nodes and edges represent dependencies between resources. Hence, it might not be connected. This graph is called *resource graph*. Figure 3 shows an example for three resources u, i and f: i and f are independent from one another but both depend on u, which does not depend on any other resource in this configuration.

The authors define a language called *FS*. "The FS language [...] is a simple imperative language of programs that manipulate the file system." FS operates on a virtual file system (σ) which maps paths (p) to their file contents; a path can either be a file's content or a directory. Resources are denoted as expressions in FS, which take a file system and yield either a new file system (i.e. the state reached after

[11]http://inspec.io

[12]https://github.com/citac/citac

[13]https://citac.github.io/eval

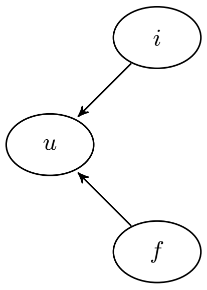


Figure 3: Resource graph with three resources, with two resources depending on the third.

running the expression) or an error. Multiple expressions are defined with FS: for instance, `mkdir(p)` and `creat(p)` for creation of directories and files respectively. `rm(p)` for deleting files and `cp(p1, p2)` for copying files. Also, sequencing and conditionals are defined and behave the usual way. Furthermore, predicates for checking the presence of a file/directory are defined as well as simple boolean operation (disjunction, conjunction and negation).

A compilation function C maps resources to FS expressions. This function is not further defined but examples for how C maps resources to FS expressions are given for files, SSH keys and package installation. For instance, for package resources, depending on the OS the package manager is invoke to download the corresponding package. Then from this package, a FS program is generated which creates the file hierarchy that the package would create on the system. The authors also mention, that several other resources are modelled like those for users, groups and services. It is also noted that C can be extended to map more resources to FS expressions if needed.

Now that all resources can be mapped to an FS expression and with the information from the resource graph, test for determinism of the configuration can be done. For this, all resources are mapped to FS expressions which are then executed in all possible sequences, in respect to the vertices between the corresponding resources.

In the end, FS programs are encoded as logical formulas, which are then solved with an SMT solver in order to determine the determinism of the configuration. In this paper, determinism of a configuration is defined as follows: when executing all possible sequences on the same σ and all of these executions yield the same end file system or error, then this configuration is deterministic.

Discussing this procedure would exceed the scope of this paper, but in short their approach has three major steps:

1. The number of FS programs are reduced by analysing if path p accessed by resource r is also by another resource. If there is no such resource then r is irrelevant for determinism-check and hence will not be checked.

2. With a commutativity check for resources is done to further reduce the number of FS programs: if the resources r_1 and r_2 yield the same file system in any order they are executed, then they are commutativity and an arbitrary execution order is chosen.

3. The final step is then to encode the remaining FS programs into decidable logical formulas which are then solved by an SMT solver to determine determinism of the configuration.

With this approach, idempotence (as well as convergence) can be tested by running a resource or the whole configuration successively. Since no running system was changed during the testing, this is testing during pre-convergence.

Their evaluation of Rehearsal includes the benchmark of 13 publicly available Puppet configurations (including configurations for installing and setup of nginx and an IRC server) as well as a synthetic benchmark.

During the test of the public configuration, Puppet tested their determinism within a few seconds each. As a result, the found a few non-deterministic configurations among these configurations as well as other bugs like missing dependency declarations.

For the synthetic benchmark, they defined multiple resources accessing the same file (which bypasses the commutativity check). This resulted in higher execution time since Rehearsal has more path so solve. The authors argue that this problem is irrelevant since Puppet would not allow such configurations.

4. DISCUSSION

While many types of testing are already covered for Infrastructure as Code, to our knowledge, some important testing techniques (see section 2.1) from software engineering are yet to be researched.

We now want to discuss testing techniques which, to our knowledge, are currently not covered in testing Infrastructure as Code.

Destructive testing. Running such destructive testing during the application of the configuration could test if the IaC tool is able to recover from this faulty state into the desired state.

This type of testing could be archive by running the configuration within a virtual environment and alter the environment during runtime to get into a faulty state.

A/B testing. It is important that a configuration does not assume a specific initial state of the system because this would contradict with the important feature of IaC being portability.

This type of testing could also be archive by running the configuration within a virtual environment. For A/B testing, the configuration would run multiple times, each time with a different initial state (e.g. different pre-installed packages or users).

Compatibility testing. Some resources are not independent of the underlying operation system but using compatibility testing would enforce avoid such resources and make the configuration more portable, even to other operation systems.

Compatibility testing would be very similar to A/B testing, with the difference that the initial states of the different runs are different operating systems (i.e. different Linux distributions).

5. CONCLUSION

In this paper we gave an overview of current approaches for testing Infrastructure as Code, both in practice as well as in current research. We showed testing techniques which are suitable for IaC and showed currently used tools which apply some of these techniques. Furthermore, current research shows that the field of testing IaC is not yet exhausted, as we also mentioned in our discussion. We then showed which testing techniques could potentially be used and gave examples on how to archive these.

Further work is needed in order to analyse these techniques in context of testing Infrastructure as Code.

6. REFERENCES

[1] A. Abran, J. Moore, P. Bourque, R. Dupuis, and L. Tripp. Software engineering body of knowledge. *IEEE Computer Society, Angela Burgess*, 2004.

[2] Ansible. ansible.com. `https://www.ansible.com/`, 2016. Retrieved December 19, 2016.

[3] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1021–1028. ACM, 2005.

[4] Chef. chef.io. `https://www.chef.io/`, 2016. Retrieved December 03, 2016.

[5] O. Hanappi, W. Hummer, and S. Dustdar. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 328–343, New York, NY, USA, 2016. ACM.

[6] M. Httermann. *DevOps for developers.* Apress, 2012.

[7] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam. *Testing Idempotence for Infrastructure as Code*, pages 368–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] R. Kohavi and R. Longbotham. Online controlled experiments and a/b tests. *Encyclopedia of machine learning and data mining*, pages 1–11, 2015.

[9] Kubernetes. kubernetes.io. `https://kubernetes.io/`, 2017. Retrieved January 03, 2017.

[10] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, July 2006.

[11] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt. verlag, 2013.

[12] K. Morris. *Infrastructure as Code.* O'Reilly Media, Inc, Sebastopol, CA, 2016.

[13] S. Nelson-Smith. *Test-Driven Infrastructure with Chef.* O'Reilly Media, Inc, Sebastopol, CA, 2014.

[14] Puppet. puppet.com. `https://puppet.com/`, 2016. Retrieved December 19, 2016.

[15] R. Shambaugh, A. Weiss, and A. Guha. Rehearsal: A configuration verification tool for puppet. *SIGPLAN Not.*, 51(6):416–430, June 2016.

[16] B. Siebert, M. van Eekelen, and J. Visser. *Evaluating the testing quality of software defined infrastructures.* PhD thesis, Thesis, Radboud University Nijmegen, 2014.

[17] D. Swarm. docker.com/products/docker-swarm. `https://www.docker.com/products/docker-swarm`, 2017. Retrieved January 03, 2017.

[18] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *1993 Conference on Software Maintenance*, pages 302–310, Sep 1993.

[19] S. Wastie. The real cost of downtime, the real potential of devops. `https://blog.appdynamics.com/engineering/idc-devops-cost-downtime`, 2015. Retrieved January 03, 2017.

[20] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 63–74, New York, NY, USA, 2008. ACM.

A Study of Cost-Benefit Analysis of Technical Debt

Radu Coanda
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
radu.coanda@rwth-aachen.de

Muhammad Firdaus Harun
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
firdaus.harun@swc.rwth-aachen.de

ABSTRACT

Since Ward Cunningham introduced the term of technical debt in his report at the OOPSLA'92 conference, the metaphor has spread across the community, helping translate technical decisions into financial ones. In such, business models for managing technical decisions have risen with it. This research work overviews a number of papers that present, as a decision model, the cost-benefit analysis, and discusses possible parameters for such a model, as well as the benefits their results would bring to the decision-making process.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*cost-benefit analysis, technical debt prioritisation, decision-making*

Keywords

cost benefit analysis, technical debt prioritization, decision-making

1. INTRODUCTION

The term technical debt(TD) was first used by Ward Cunningham in 1992. The TD metaphor describes a situation in which developers accept compromises which produce short-term benefits in exchange for the long-term health of the software. In practical terms, one could compromise in one dimension (e.g. maintainability) to meet an urgent demand in another dimension (e.g. delivering a release on time) [2]. This metaphor is related to immature, incomplete or inadequate artefacts in the software development cycle that cause higher costs and lower quality.

The usage of TD, increased in the software development community to help with the management of software projects as well as to help communicate return on investment (ROI) to non-technical stakeholders for better decision-making [2].

In literature and software industry different terms and properties are attributed to TD, including TD item, principal, interest amount and probability:

- **TD item** - The TD associated with a system is composed of TD items, where each item represents a technical problem the system has and can be attributed a principal and an interest amount.
- **Principal** - Principal refers to the cost one has to pay for fixing the TD and can be defined as the cost of repairing quality issues in software systems to achieve an ideal quality level. In practical terms, an ideal quality level represents the highest achievable level of quality defined in a software quality model adopted by an organization [7].
- **Interest** - Interest refers to the cost one will have to pay for not fixing the TD item. For example, the lack of proper documentation might lead to high maintenance effort later on.
- **Interest probability** - The obvious distinction between interest and principal is that the former is not certain, but has probability attached to its occurrence, which changes over time.

In a perfect world TD would be something unheard of, as engineers like to build perfect software. Though, if we take into account the fact that most of the software that is written lies under resource constraints, time-to-market, target audience, etc., TD emerges as the trade-off the software engineers do to deliver the software while meeting all these constraints. Thus, for a team not to be overwhelmed by the amount of debt they have to pay and for the project not to go "bankrupt", management for paying TD is required.

The task of managing TD consists of two sub-activities: first, identifying the different types of TD and second, determining the economical value and impact for removing TD items or not the. Tools, methods and different approaches regarding the identification of TD have already been thematized in a large body of literature [14]. Prioritizing TD, though, is a less charted topic, yet there is already research being pursued in this area. By countinuing the economical metaphor, models from the financial sector, like *Portfolio Approach*, as well as the *Options Approach*, have found their way in the software development community[11].

For the purpose of this paper, we will take on the decision-making perspective as filtered through the cost-benefit analysis.

1.1 The Cost-Benefit Model

In economical literature a cost-benefit analysis provides an economic framework to evaluate the viability of a proposed or operating project. It can be defined as the systematic gathering of technical and financial data about a given business situation or function. Information gathered and analysed through this method assists decision-making about resource allocation and selecting the appropriate alternative, by comparing the "with" and "without" situations [3].

A cost-benefit model in the TD management context, thus approximates the interest as well as the principal as an aid to the decision-making process. In a formal sense the cost-benefit analysis is a systematic approach to estimate the strengths and weaknesses of the alternatives; it is used to determine options that provide the best approach to achieve benefits while preserving savings.

Several papers on the topic of prioritizing TD have been produced. In section 2 we will proceed by looking at the challenges that arise from translating technical debt into economical consequences, followed by section 3 where we are going to look over two subjectively chosen research works, the model by Zazworka et al. and the SIG model of the Software Improvement Group (SIG) Amsterdam presented by Nugroho et al. The models have been chosen based on the completeness of the model, as well as the object of their study. At the end of each overview a discussion will take place to review all the challenges being tackled by the model. Two alternative estimation models will be presented as followup, at the end of section 3 [7][16]. Finally we will discuss two additional directions of possible future research work and draw the conclusion.

2. CHALLENGES

For the purpose of this paper we will first look at the major difficulties one might expect when trying to manage and prioritize TD items. We will take on the perspective proposed by Falessi et al. in their paper regarding a TD managing tool, in order to loook at the challenges one practitioner might incur [4]. The study was done at a CMMI Maturity level 5 company. Capability Maturity Model Integration (CMMI), is a process level improvement training and appraisal program, developed at Carnegie Mellon University. A maturity level is a well-defined evolutionary plateau toward achieving a mature software process. A company with a level 5, the highest level, focuses on continually improving process performance through both incremental and innovative technological improvements.

2.1 Principal

Since a software product is written by the human hand, a considerable amount of effort is required for each necessary fix, and so the first challenge arises.

C1: Inadequacy of single values.

As effort is specific to each individual, assigning one value (cost) to one's programming activity for fixing a problem is subjective.

Furthermore, in their overview of the practitioners at one CMMI maturity level 5 firm, Falessi et al. understood that developers tend to think in ranges, rather than in single values, e.g. best-case, worst-case. Yet still the approximation error resulted on average of an 80% over- or underapproximation of the correct value [4].

2.2 Interest

The interest amount is dependant on the TD probability of occuring [4]. Under this consideration Falessi et al. reasons the following challenges:

C2: Distance from the economic domain.

TD relates to a problem in the technical domain which can be very far away from the economical domain. For instance, despite the reasonable expectation that little documentation leads to high maintenance effort, it is very hard to estimate the cost (interest) of not documenting a design in UML, a defect in JIRA or commenting a JAVA class [4].

C3: Interaction effect.

The cost of a TD item may increase in the presence of other TD items. For example "bad documentation" and "uncommented code" increase the interest when both TD items are present at the same time [4].

C4: Non-linearity and limits.

TD can change over time, in such it can happen that the interest might increase exponentially instead of linearly[4].

C5: Multiple interest.

One TD can negatively impact several software quality aspects at the same time. For instance, an "uncommented code" has a negative impact on both maintainability and reliability, whereas each aspect can have its own amount of interest and probability[4].

C6: Unavailability of historical data.

Historical data is required for estimating the amount of interest for a specific organization. For instance, by adopting a configuration management system and analysing source code repositories data we can see the extent to which a component with high coupling and cohesion is less maintainable than other components [4].

C7: Interest probability of occurrence as probability of events.

The probability of the interest occurring in the future depends on the existence of future "events" [4].

Having comprised a subjective list of requirements for a TD management model, we are further going to look at how these requirements are being met by the following models.

3. COST-BENEFIT MODELING

For the TD metaphor to be useful, its constructs must be measurable or at least estimable from measurable elements of software. This is especially important for a functional framework, used for decision-making, to work.

The models used in this research work, tackle the aforementioned challenges in different ways.

3.1 Cost/Benefits of single TD items

In his 2011 research work, Zazworka et al. introduce a decision model for estimating the cost and impact for refactoring a single type of TD item in a software project, in particular the code smell called ”God Class” [13].

3.1.1 *God Class and Cost*

The god class code smell characterizes classes that overcentralize functionality and have multiple responsibilities in a system. These classes are more concerned with the data of other classes than their own.

God classes can be identified using Marinescu’s detection strategy that uses three software metrics, namely Weighted Method Count (WMC), Tight Class Cohesion (TCC) and Access to Foreign Data (ATFD)[8]. Passing a certain threshold on each of these metrics, results in the class being chosen as a candidate for a ”god class”.

A ranking of the resulting classes is being done afterwards, based on the distance of each class from the threshold for each individual metric. The overall ranking of the classes results from all three metrics summed up, as seen in figure 1.

God Class Name	**WMC (>46)**		**TCC (<0.33)**		**ATFD (>5)**		**Overall Score and Rank**	
	Value	Rank	Value	Rank	Value	Rank	Rank Sum	**Rank**
GodClass1|	49	3	0.0	8	20	6	17	6
GodClass2	87	8	0.005	7	28	7	22	7
GodClass3	107	9	0.0	8	28	7	24	9
GodClass4	69	7	0.026	6	34	9	22	7
GodClass5	49	3	0.065	5	9	3	11	3
GodClass6	60	5	0.177	4	19	4	13	4
GodClass7	47	1	0.219	1	7	1	3	1
GodClass8	48	2	0.199	2	7	1	5	2
GodClass9	61	6	0.192	3	19	4	13	4

Figure 1: Refactoring effort ranking based on detection strategy metrics.

3.1.2 *Benefits*

If the first dimension of the approximation method measures how costly it would be to refactor the class, then the second dimension of the approximation method determines the impact the class has on a set of quality characteristics. The two characteristics considered in the example are [16]:

- *Correctness*, represented by the defect likelihood and
- *Maintainability*, represented by change likelihood of a god class.

Correctness of a class can be estimated by the defect likelihood measure, similar to the one defined by Schumacher et al. [10]. For a god class one can compute how many defect fixes affected this god class by mining the code repository and issue tracker. More specifically, one will account for the time the class was a god class (e.g. from May to September), then count the number of defects that lead to fixes in the god class in this time period, and divide by the number of all defects that were fixed in this time period. The higher the resulting value, the more likely a defect will manifest in the god class, e.g. a likelihood of 0.5 would indicate that every second defect fix leads to changes in this god class [16].

Maintainability can be estimated by investigating how often a class is changed. For this the change likelihood, as defined in[10], indicates how likely a class is to be modified when a change to the software is made. For example, a change likelihood value of 0.1 shows that the class was, on average, modified with every 10th change (i.e. revision) to the software [16].

God Class Name	**Change Likelihood**		**Defect Likelihood**		**Overall Score and Rank**	
	Value	Rank	Value	Rank	Rank Sum	**Rank**
GodClass1	0.016	1	0.0	1	2	1
GodClass2	0.097	8	0.0	1	9	4
GodClass3	0.102	9	0.029	5	14	9
GodClass4	0.068	7	0.177	6	13	7
GodClass5	0.040	3	0.0	1	4	3
GodClass6	0.0455	4	0.133	7	11	5
GodClass7	0.0458	5	0.133	7	12	6
GodClass8	0.052	6	0.133	7	13	7
GodClass9	0.027	2	0.0	1	3	2

Figure 2: Software quality characteristic ranking based on change and defect likelihood.

Both metrics, change and defect likelihood, can be calculated for each of the god classes and the results can be ranked using an approach similar to that described in the previous section. Ranking can be seen for our example in figure 2.

3.1.3 *Model*

In the model of Zazworka et al. the findings are then plotted in a cost-benefit matrix, like in figure 3. The two axes correspond to the two ranking dimensions: refactoring effort and quality impact.

Looking at the matrix one can make following observations: classes that fall on the diagonal, tend to have balanced effort/impact ranking characteristics. Classes above the diagonal are the most promising to refactor, since the impact is ranked higher than the effort. For example GodClass7 and GodClass8 are potentially inexpensive to refactor and have a relatively high negative impact on software quality. On the contrary, classes below the diagonal tend to have little impact and high refactoring cost. This debt is likely to have low interest (i.e. low impact on quality) and high value. For

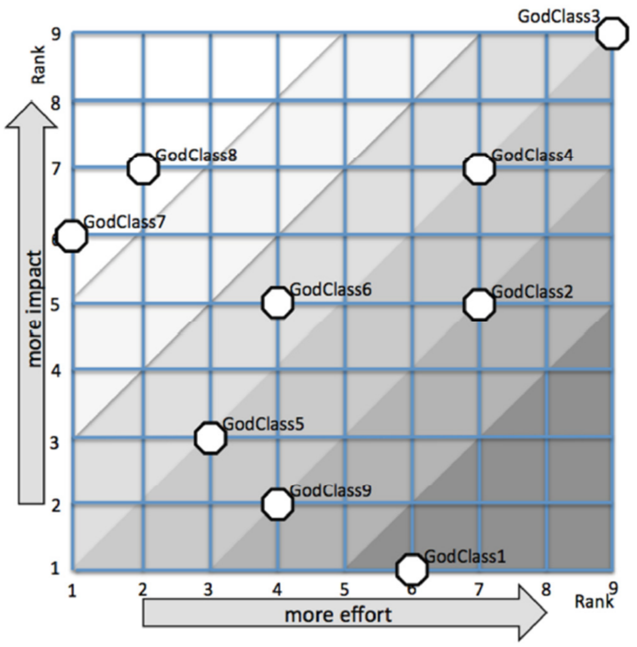


Figure 3: Design Debt Cost Benefit Matrix for god classes.

example, GodClass1 is likely to be one of the more expensive ones to fix, but does show only little negative impact. Fixing this debt can be deferred to a later point in time [16].

Zazworka et al. analysis was made based on god classes. Such an analysis isn't restricted to only this code smell. In the same way, by picking for the effort dimension relevant metrics for finding the TD items and relevant quality characteristics for the second dimension, one can construct the same cost-benefit matrix to decide on other types of TD.

With this strategy the following challenges are being addressed: *C1*, although there hasn't been given an absolute value for a TD item, the ranking that was done for all god classes, resulted in a relative value being assigned to each class. *C2*, by taking into account the distance from an optimal quality level, based on characteristics such as *correctness* and *maintainability*, a non-technical stakeholder is able to take decisions based on this analysis. *C5*, using multiple software quality characteristics, one can choose which type of interest is relevant to him.

3.2 System-wide decision-making

TD represents the cost of improving the quality of software to the level that is deemed ideal. Based on this observation, Nugroho et al. have developed at the Software Improvement Group Amsterdam (SIG) a model for estimating the cost as well as the benefits of raising the overall quality of a software product. In the following we will look at the general model for their approximation, but also understand the resulting data after having used this model in an example [7].

3.2.1 SIG Quality Level

To improve a systems overall quality level, an "ideal" level of quality has to be first determined. To this aim, SIGs's software quality assessment method is used.

This method is developed as a layered model for measuring and rating the technical quality of a software system in terms of the quality characteristics of ISO/IEC 9126 [1].

In the first layer, source code analysis is performed to collect measurement data about the software system. The analysis involves well-known metrics such as LOC (LOC), code duplication, McCabe's cyclomatic complexity, parameter counts, and dependency counts. These metrics are collected at the low level system elements such as lines, units (e.g., methods or functions), and modules (e.g., files or classes)[7]. For more details on how the metrics are measured please refer to [5].

Subsequently, these metrics are mapped onto ratings for properties at the level of the entire software product, such as volume, duplication, and unit complexity. These ratings take values in the interval between 0.5 and 5.5, which can be rounded to an entire number of stars between one and five. This constitutes a unit-less ordinal scale that facilitates communication and comparison of quality results at the level of entire software products[7].

3.2.2 Cost

As in the previous model, the first dimension is the estimated cost for fixing the system, or in other words the principal of the TD. When estimating the cost of repair, what is being estimated is actually the *repair effort* (RE) spent to perform the repair work. If repair work is performed to increase quality to the ideal level, then RE will represent the amount of technical debt. Nugroho et al. define RE as the product of *rework fraction*(RF) and *rebuild value*(RV) [7]:

$$RE = RF \times RV$$

Rework Fraction

In their model Nugroho et al. estimate the RF based on SIG's software maintainability rating. The estimations for all ratings can be seen in figure 4. The figures represent the percentage of LOC that will need to be changed in order to raise the quality by one level. For example, to improve a system from 2-star to 3-star rating would require changes as much as 40% of the code. For further detail please refer to [7].

Target/Source	1-star	2-star	3-star	4-star	5-star
1-star					
2-star	60%				
3-star	100%	40%			
4-star	135%	75%	35%		
5-star	175%	115%	75%	40%	

Figure 4: Estimated Rework Fraction.

Rebuild Value

RV is an estimation of effort (in man-months) that needs to be spent to rebuild a system using a particular technology. This is determined by the following formula:

$$RV = SS \times TF$$

where:

- *System size*(SS) - represents the total size of a system measured in LOC. Alternatively, SS can be measured using functional size (i.e. Function Points).

- *Technology factor*(TF) - represents language productivity factor. This factor provides a conversion from source code statement to effort (i.e. man-months per source statement) through 'backfiring'. Please refer to [6],[9] for further discussion on the subject.

3.2.3 SIG Quality Model Benefits

For the second dimension of the model we continue with Nugroho et al. 5-star rating approach. Thus, we reason that the interest is the difference between maintenance effort spent at the 5-star level and any of the lower quality levels [7]. With this observation in mind the Nugroho et al. concluded the following formula for estimating the *maintenance effort* (ME) at various levels:

$$ME = \frac{MF \times RV}{QF}$$

The ME is thus a function of the *Maintenance Fraction* (MF), *Rebuild Value* (RV), and *Quality Factor* (QF). The three variables are described as follows:

- QF - is the factor used to describe the level of quality. The higher the level of quality, the less effort spent on maintenance. For examples motivating these findings see [7]. QF is determined using the following formula:

  $$QF = 2^{\frac{QualityLevel-3}{2}}$$

  The formula gives the following factors for quality level from 1-star to 5-star respectively: 0.5, 0.7, 1.0, 1.4, 2.0.

- MF - represents the amount of maintenance effort spent on a system in a yearly basis, measured as a percentage of LOC that is estimated to change (added, modified, or deleted) yearly, due to maintenance. It is based on historical acquaintance data. The authors made the assumption based on their data, that yearly changes in code takesapproximately 15% of the developer's time.

- RV - the rebuild value is calculated as described in Section 3.2.2 Cost.

A real-world example for this formula is then calculated by the authors, on a 2 star system under supervision.

$$ME_{2*} = \frac{15\% \times 186.7}{0.7} = 40$$
$$ME_{5*} = \frac{15\% \times 186.7}{2} = 14$$

Reviewing the results of the above calculation, Nugroho et al. concluded that for this project with a 193 man-months worth of debt, 26 man-months (40-14) of extra effort would be spent on maintenance on a yearly basis. This would translate in economical terms into a TD worth \$1,608,333 and an interest of \$216,666, under the assumption that the yearly cost of a maintenance staff is set at \$100,000 [7].

3.2.4 Model

For the overall assertion of the model, a case study was undertaken. The case study is a system that was recently assessed by SIG. Its main functionality is designing transportation infrastructure. When the assessment was done the system was 18 years old. The system was developed using various technologies including Java, C, C#, C++, PHP. The size of the whole system is 760,000 LOC [7].

The result of the quality assessment revealed that system has a 3-star quality level. Figure 5 provides the results of TD calculation of SIG case study system for the following 10 years. The table provides two scenarios: 1) keeping the quality at 3-star level; 2) improving the quality level to 4-star. In the last row of the table, financial projections of ROI for investing in quality improvement to the 4-star level are provided.

Table 5 shows that the required RE to improve system quality to 4-star is 48 man-months (\$400,000). The table also shows the debt and interest need to be paid over 10 years. It is shown that the amount of debt and interest over time at the 4-star level is nearly half that of 3-star. By calculating the saving on interest as a result of moving to 4-star, and also taking into account the repair effort, the return on investment (ROI) can be determined. Table 5 shows that the repair effort invested to improve the quality level to 4-star will pay back in terms of positive ROI of 15% in seven years.

Although Nugroho et al. estimation model focuses on whole systems, the analysis could be constrained to tackle particular TD items as well.

By benefiting from the code repositories under their supervision, Nugroho et al. developed estimations by mining their historical data, which in turn leads to the assumption of having historical data as an asset. The same argument leads, though, to the conclusion that the challenges *C1, C2, C3, C5, C7* are being indirectly, but nonetheless tackled by this estimation model.

Given that the model is based entirely on estimations related to historical data analysis, the results could be not at all relevant to the current system.

3.3 Alternative models

Having overviewed the aforementioned cost-benefit models, one can make the following observation: cost-benefit analysis is divided into two dimensions. In the case of TD management a cost-benefit analysis assumes the principal for the dimension of the cost and the interest for the dimension of the benefits. We can reason, that it is then possible to interchange different estimation strategies for one dimension, while still obtaining a worthwhile result.

We will thus look over one alternative for each of the dimensions of a cost-benefit model. We will start by going over an alternative for the principal estimation, by using the CAST estimation model and then overview one other models for the interest estimation.

3.3.1 CAST Model

In their exploration of TD management at CAST Software, Curtis et al. introduced a new approach to TD principal estimation and TD prioritisation, by focusing only on the should-fix violation. Curtis et al. characterize these as violations of good architectural or coding practice (hereafter

		Year									
		1	2	3	4	5	6	7	8	9	10
3-star	Debt*	102.9	109.3	116.1	123.4	131.1	139.2	147.9	157.2	167.0	177.4
	Interest*	11.1	12.5	14.0	15.6	17.3	19.0	20.9	22.9	25.0	27.3
	Maintenance effort*	22.3	23.7	25.2	26.7	28.4	30.2	32.1	34.1	36.2	38.4
	Required resources **	1.9	2.0	2.1	2.2	2.4	2.5	2.7	2.8	3.0	3.2
4-star	Debt*	54.9	58.3	61.9	65.8	69.9	74.3	78.9	83.8	89.1	94.6
	Interest*	4.6	5.6	6.6	7.8	8.9	10.2	11.5	12.9	14.4	16.0
	Maintenance effort*	15.8	16.7	17.8	18.9	20.1	21.3	22.7	24.1	25.6	27.2
	Required resources**	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2.1	2.3
	Repair Effort*	48.0									
	ROI	-86.4%	-72.0%	-56.6%	-40.3%	-23.0%	-4.6%	15.0%	35.8%	57.8%	81.3%

* in man-months; ** in man-years;

Figure 5: Technical debt of SIG case study system on a 10-years horizon

referred to simply as "violations") known to have an unacceptable probability of contributing to severe operational problems (outages, security breaches, data corruption, and so on) or of contributing to high costs of ownership, such as excessive effort to implement changes.

As software is a product of the mind, that is being handcrafted, the main resource going into the software is manpower or more clearly effort. The measure of effort should take into account the the employee's hourly salary, as well as providing him with the necessary tools to perform his trade optimally.

Thus, we reason the following parameters of importance: number of TD items in an application, the hours to fix each TD item, and the cost of labor at an hourly rate. Using these three variables Curtis et al. came up with the following equation for estimating the TD-principal [15]:

$$\begin{gathered}\text{TD-principal} = \\ ((\Sigma \text{ high-severity violations}) \times (\text{percentage to be fixed}) \times \\ (\text{average hours needed to fix}) \times (\text{US\$ per hour})) + \\ ((\Sigma \text{ medium severity violations}) \times (\text{percentage to be fixed}) \times \\ (\text{average hours needed to fix}) \times (\text{US\$ per hour})) + \\ ((\Sigma \text{ low-severity violations}) \times (\text{percentage to be fixed}) \times \\ (\text{average hours needed to fix}) \times (\text{US\$ per hour}))\end{gathered}$$

3.3.2 Maintenance Behaviour

Although such models provide the means to estimate debt, it may be difficult to justify reducing TD without detailed information about the impact of the debt on developer's day-to-day maintenance activities. Under the assumption, that most developer effort during software maintenance is spent on program comprehension activities, such as reading and navigating code, Singh et al. reason, that understanding the impact of structural-quality-related debt on code comprehension is of critical importance[12].

By using a plug-in, called *Blaze*, for Visual Studio(VS) the authors were able to monitor and record the behaviour of developers during maintenance cycles, e.g. code navigation actions as well as edit actions, in a log. By analysing the log, Singh et al. understood the class relationships and quantified the effort spent by a developer to comprehend individual porgram elements while completing a change task. In combination with the code maintainability measurements presented by Nugroho [7], the comprehension effort data, can provide evidence of how TD impacts developer-code-comprehension effort and thus update the interest continuously[12].

With the method Singh et al. developed, one could update the interest of a TD item in real-time, therefore help actively prioritize TD and thus possibly avoid situations in which the debt reaches a point at which it has a substantial impact on the progress or the cost. Such an aid would address specifically challenge *C4* by having the TD under continuous supervision.

4. CONCLUSION

TD is an overwhelming concept and managing TD requires dealing with heterogeneous stakeholders, conflicting needs, and a considerable level of uncertainty, factors that are hard to account for[4]. The presented models try to resolve at least the technical challenges and guide, even the less tech-centered stakeholders, in the the decision-making process for a software project.

Two considerations for further studying of cost-benefit models have come up during the research for this paper:

- *Overhead cost* - a new dimension one could take into their calculation could be the cost one might incur by actively tracking the debt in the project. Overcomplicated tools tend to be more of an inconvenience than an aid.

- *Time* - another dimension to consider could be the time and how the debt progresses in this dimension. Not only in the past but also how paying up the debt might behave in the future. This has been lightly touched upon in SIG's model, by extending the prediction to a 10-years horizon. This might allow developers to pay up a certain amount of the debt for one TD item and then switch to another one.

The cost-benefit model (and in particular the models presented in this research work) is not a silver bullet approach, as we still have not addressed all the challenges with any of the models. TD management is an area in which the knowledge and experience of developers is still required for the decision-making process. The models are as mentioned in the introduction only an aid for them.

Further research is needed, though, the work already done, as presented in section 3, proves that a solid base is already being build. There still has to come an approach to solve more of the challenges expressed in section 2.

5. REFERENCES

[1] I. 9126. Software engineeringâĂŤ product quality. *IntâĂŹl Org. for Standardization*, 2001.

[2] W. Cunningham. The wycash portfolio management system. *Addendum to Proc. Object-Oriented Programming Systems, Languages, and Applications, ACM*, pages 29–30, 1992.

[3] R. David, P. Ngulube, and A. Dube. A cost-benefit analysis of document management strategies used at a financial institution in zimbabwe: A case study. *SA Journal of Information Management 15(2)*, 2013.

[4] D. Falessi, M. A. Shaw, K. Mullen, and M. Stein. Practical considerations, challenges, and requirements of tool-support for managing technical debt. *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 16–19, May 2013.

[5] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, Sept 2007.

[6] C. Jones. Backfiring: converting lines of code to function points. *Computer Volume: 28, Issue: 11*, pages 87 – 88, August 2002.

[7] A. Nugroho, J. Visser, and T. Kuipers. An empirical model of technical debt and interest. *MTD '11 Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 1–8, May 2011.

[8] M. R. Detection strategies: Metrics-based rules for detecting design flaws. *In Proceedings of the 20th IEEE international Conference on Software Maintenance. ICSM. IEEE Computer Society, Washington, DC*, pages 350–359, September 2004.

[9] S. P. Research. Llc. spr programming languages table. *Ver. PLT2007c*, December 2007.

[10] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw. Building empirical support for automated code smell detection. *In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10). ACM, New York, NY, USA*, 2010.

[11] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. VetrÃš. Using technical debt data in decision making: Potential decision approaches. *MTD '12 Proceedings of the Third International Workshop on Managing Technical Debt*, pages 45–48, June 2012.

[12] V. Singh, W. Snipes, and N. A. Kraft. A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension. *2014 6th IEEE International Workshop on Managing Technical Debt*, pages 27–30, 2014.

[13] W. Snipes, B. Robinson, Y. Guo, and C. Seaman. Defining the decision factors for managing defects: a technical debt perspective. *MTD '12 Proceedings of the Third International Workshop on Managing Technical Debt*, pages 54–60, June 2012.

[14] C. Sterling. *Managing Software Debt: Building for Inevitable Change.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2010.

[15] A. Szynkarski, J. Sappidi, and B. Curtis. Estimating the principal of an application's technical debt. *IEEE Software, vol. 29, no.*, pages 34–42, November 2012.

[16] N. Zazworka, C. Seaman, and F. Shull. Prioritizing design debt investment opportunities. *MTD '11 Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 39–42, May 2011.

Facing *Synthetic Workload Generation* as part of Performance Testing – a tools approach

Marco Moscher
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
marco.moscher@rwth-aachen.de

Konrad Fögen
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
foegen@swc.rwth-aachen.de

ABSTRACT

Nowadays performance engineering does not only face basic KPI analysis nor only responsibility testing during a given workload. Since the *System Under Test* (SUT) does get more data complex, a performance engineer is committed to take simple data generation into account. While creating small and valuable amount of test data is easily achievable by hand, it is impossible for large test cases, e.g. 100 units or above. Though, without good tooling, this process could evolve as a very time consuming task during a representative test.

To elaborate existing opportunities and methods to address workload generation, this paper compares the following performance testing approaches focusing data generation possibilities. The techniques *Capture and Replay* (CR) and *Model-Based Testing* (MBT) are explained by a realization of each. On the one hand *Microsoft Visual Studio* [13] is mentioned to point out CR approach opportunities. On the other hand *PLeTsPerf* [16] is chosen as representative approach to MBT.

It becomes apparent that both approaches preclude the process of data generation, which indicates that an additional (approach independent) possibility is required to compile good, realistic test data. For that reason this paper moreover focuses the creation of synthetic workload using a tools-based approach for a small, fictional System-Under-Test and a test scenario.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

performance testing, performance engineering, model-based testing, capture and replay testing, data generation, synthetic workload

1. INTRODUCTION

Performance testing itself appertains to the *testing* phase during software development, one of the most important and time consuming phases. Beside well established *Unit*, *Integration*, *System* and *Acceptance* testing phases, performance testing is often neglected since it is a highly specialized task [7]. This accrues through much required knowledge regarding the System-Under-Test (SUT), e.g. its usage profile and infrastructure where it will operate on [16], and lacking expertise on performance testing tools [7]. Nevertheless, it is a crucial task, because high response times can have a negative impact on customer satisfaction [1].

The basic idea of performance testing is to validate the SUT among a synthetic workload – also known as *load-* or *stress-* test [16]. During this test various KPI's like responsiveness, stability and resource utilization are monitored and evaluated in a controlled environment. A synthetic workload is applied to simulate the expected load under operational use as closely as possible [1]. Therefore, a good method on synthetic workload generation is essential to conclude reliable results from the test results. Through many and smart possibility of data collections (*Internet of Things, Big Data*), software system get more data intensive, whereby automatic workload generation for performance testing needs to be considered strongly to provide *enough* and *plausible* data.

In this paper we will present and compare established approaches towards performance testing. We will outline the aspects of data generation while realizing the data derivation task of a test scenario for a fictive SUT theoretically. The main contribution of this work is to clarify possibilities and the different proceedings on synthetic workload generation when adopting a particular approach. Therefore, we answer the following research question (RQ) in general, by examining two key questions (KQ) on each approach.

RQ Is the retainment of synthetic workload considered as part (subtask) of performance testing, i.e. no additional effort on other field of software testing needed, or does a performance engineer faces additional tasks?

KQ-1 Is the approach itself capable of generating test data?

KQ-2 How to obtain synthetic workload according to the utilized approach?

The paper is structured as follows: In Section 2 we give overview on the related work, followed by a short introduction to performance testing approaches in Section 3. In the principal part, Section 4, the fictive SUT and its scenario are presented to set the basic context. Thereupon we characterize both of the previously introduced approaches regarding data generation while answering the previous presented key questions. Finally, in Section 5 and 6, we discuss and present our outcomes.

2. RELATED WORK

Even though much literature and tool-approaches on the topic of performance testing exists, elaborations focusing one crucial part[1] - namely synthetic workload generation - are rare. One will recognize, that the main focus in the current field of performance engineering is set on automatic and reusable generation of test artifacts like scripts and models.

M. Bernardino *et al.* [4, 17] present two different empirical studies in the field of performance testing. The first one published in September 2014, *Evaluating Capture and Replay and Model-based Performance Testing Tools: An Empirical Comparisons* [17], focuses available tools distinguished by their used approach (CR-based or MBT-based). Different tools, applying one of the mentioned technique, were evaluated and compared based on effort (time spent) to create test artifacts. Their results indicates that the advantage using MBT grows significantly when the test complexity increases. The newer one published in April 2016, *Performance Testing Modeling: an empirical evaluation of DSL and UML-based approaches* [4], was motivated by the lack of a (known) standard to represent performance testing information. Though the effort and suitability – as well as in the previous study – for modeling performance testing with UML- and DSL-models was evaluated. Their results indicate that applying DSL is less effort than using UML.

As our best research yield, yet no studies nor empirical evaluations exists, which investigate the (un-)existing correlations between data generation and approach-based performance testing in place.

3. BACKGROUND

In the last decades two distinct approaches in the field of performance testing come into being. A variety of tools have been developed, which take advantage of these techniques [5], for what reason *CR* and *MBT* -techniques represent the *current-state-of-art* in this field. It is necessary to distinguish between both when focusing on central aspects like data generation during performance testing.

3.1 Approaches

Common approaches for performance testing are defined methods and proceedings for *test scripts*, *test scenarios* and *workload* creation to perform the actual test. Thus a *approach* towards performance testing does not focus the test (execution) itself, even if it might be possible with the utilized tool implementing the desired approach.

3.1.1 Capture and Replay Testing

Applying a Capture and Replay (CR) technique, the test engineer has to perform the tests manually once on the SUT (*capture*) while using an appropriate testing tool (load generator) in record mode [17]. Thereby all inputs or interactions on SUT and additionally the resulting output of the performed action are recorded. Later the *load generator* can rerun all captured tests when a new release of the SUT is deployed (*replay*) and report which ones succeeds or fails.

[1]Executing performance test against a SUT without any appropriate synthetic workload and/or runtime data is, obviously, senseless.

CR-based approaches are widely spread as a result of their simplicity and low adaption effort. The most common tools supporting a CR-mode are Apache JMeter [8], Microsoft Visual Studio [13], HP LoadRunner [12] and IBM Performance Tester [9]. As the process for data generation in a CR-approach is similar overall tools, we will exemplary introduce Visual Studio as a representative tool.

Microsoft Visual Studio is an Integrated Development Environment (IDE) developed and distributed by Microsoft. Primarily it is used as rich editor for several .NET-Framework based languages like C# or VisualBasic (VB). The software itself is available in different, feature-rich, versions as *Enterprise*, *Professional*, *Test Professional* and such like. The latter one includes support for automated test, such as performance testing on a CR-based approach. In Visual Studio the tester manually interacts with the SUT, mostly a Web-based Application, and *captures* the test scenario as described before. Moreover, the engineer has the possibility to deploy her performance test (load test) into a cloud environment (Azure, provided by Microsoft as well), to easily run test in parallel, whereby an even large load could easily be created.

3.1.2 Model-Based Testing

In contrast to a CR-based approach the *model-based testing* one is a more abstract one and requires the tester to define model representation for each test of the SUT. Therefore, one could make use of existing Unified Model Language (UML) Diagrams, Use Case (UC) Diagrams or Customer Behavior Modeling Graph (CBMG). Each test-model requires the definition of additional performance information, wherefore the chosen notation (UML, UC, CBMG) needs to provide meta models to instantiate the models. Next, the generation of scripts and scenarios is accomplished by utilizing appropriate tools. Research resulted that existing tools for this are rare. The most mentioned tools are *MBPeT* [1], *TestOptimal* [18], *PLeTsPerf* [16] and *Canpous* [5]. The two latter ones will be introduced shortly.

PLeTsPerf is a model-based performance testing tool to support the automatic generation of scenarios and scripts from application models. It was developed by M. Bernardino *et al.* [16] in collaboration with an IT company in 2015. PLeTsPerf is based on the attempt to describe the SUT with aid of two broadly used UML diagrams: *UC* and *Activity Diagram* (AC). Thereby each actor within a test scenario is represented as part of an UC Diagram and test cases are derived from AC Diagrams [16].

Canopus represents a Domain-Specific Language (DSL) for performance testing rather than a complete toolset. Hence, using Canopus the whole performance test is represented in a particular language and afterwards transformed into third-party scripts (e.g. for usage with VisualStudio or LoadRunner) or a Canopus-XML File [5].

4. DATA GENERATION

Although many tools and different approaches on performance engineering exists, test engineers still face issues when replacing or adopting these processes [17], e.g. missing knowledge on appropriated tools or methods for (synthetic) workload generation. This is enhanced due to the fact, that it is hard to find literature, thematising the test data retention regarding each approach (cf. Section 2) exist. Though we will outline the challenge of data generation in the following, separated to approach dependent and independent challenges.

4.1 Assumptions and Test Scenario

To concentrate on the retainment of synthetic workload exclusively, i.e. not discussing and explaining detailed steps towards the essential task, we set the following preconditions.

Pre-1 Test artifacts, such as model or record, exists.

Pre-2 For the SUT no base- nor runtime- data yet exists.

As both presumption are independent of the data generation process itself and *vice versa*, it is permissible to assume those. Based on this foundation we deduce an answer to the previous introduced key questions **KQ-1** and **KQ-2** (cf. Section 1) for each approach. To answer these more practically and to explain the process of synthetic workload generation we use a simple, theoretical SUT *Bidder*.

Bidder - SUT and Test Scenario.

Bidder is a small e-commerce platform where customers can register, offer products, and bid on those offered by others. It is necessary that Bidder scales well over time, whereas it should be tested with a considerable and realistic (synthetic) workload in the early stages of development. The testing scenario should be like follows: During a load of `100.000` unique registered and logged-in users, 60% (selected at random) should bid on 20% of all available products (distributed at random). To represent a standard quantity of products, each registered user should provide 5 distinct ones. All remaining users, those who are not involved into a bidding process, should either watch running auctions (20% of all) or visit other products (20% of all as well). This sums up to an additional 'default' load for the test. To conclude, the whole scenario is based on three different (user-type-based) sub-scenarios: *bidder*, *watcher* and *visitor*.

Beside the outlined test scenario (load distribution), no base- or runtime- data is provided beforehand as stated in **Pre-2**. Thus a performance engineer has to ensure that this data is provided, for which the appropriated UML Class Diagram is depicted in Figure 1.

4.2 Approach Dependent Opportunities

CR-based approach.

Given **Pre-1** we assume that all required scenarios, three in total, are recorded as a *Web Performance Test* (cf. 'Walkthrough: Recording and Running a Web Performance Test' [14]) and combined into a single *Load Test* (cf. 'Walkthrough: Creating and Running a Load Test Containing Web Performance Tests' [14]). Within this Load Test each Web Performance Test could be weighted to accomplish the challenged scenario distribution (*bidder*:60%, *watcher*:20%, *visitor*:20%). Although this (basic) test is sufficient to perform a performance test against *Bidder* using `100.000` users, which are simulated through single instances (`100.000` in parallel), the *unique user* requirement of each (simulated) is not yet fulfilled, as the current data is not synthetic enough. To amend this data to be more synthetic, i.e. as realistic

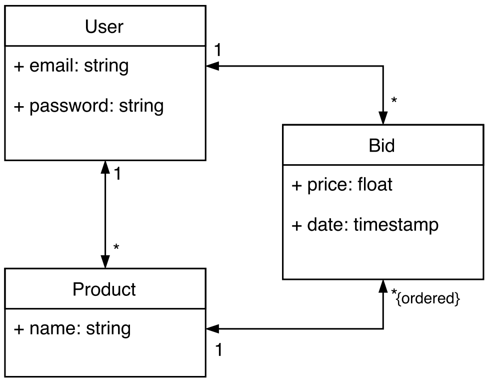


Figure 1: Class diagram illustrating all relevant Classes for *Bidder* and the relation between each.

as possible e.g. unique users, all recorded data *entered* (i.e. form data) or *clicked* (i.e. link clicking) during test, needs to be extracted and getting alterable trough *Extraction Rules* (cf. 'Using Validation and Extraction Rules in Web Performance Tests' [14]). Hereafter Visual Studio expects an external data source to map required test data to the performance test, e.g. a CSV, XML or Database (Microsoft Access, SQL, etc.) (cf. 'Add a data source to a web performance test' [14]). Thereby the performance test could be equipped with a set of unique user information to fulfill the mentioned test requirements of `100.000` unique users. It is to emphasize that the CR-based approach relies on an external data source to perform a test with *synthetic* workload – this answers **KQ-2**.

Demanded on **Pre-2** the SUT *Bidder* does not yet hold any runtime data in storage. More precisely no *User Accounts* nor *Products* exists, whereas it is not possible to create/record the claimed *Web Performance Test* as stated above. This follows a contradiction between both presumptions (**Pre-1**, **Pre-2**) when utilizing the CR-based approach only. Thus the first key question (**KQ-1**) is to be negated, since this approach is not capable of providing or generating test data itself. More precisely it is dependent to and only applicable if (at least runtime) data exists.

MBT-based approach.

"Model-Based Testing is the Automation of Test Design of Black-box Tests", defined by M. Utting *et al.* [19] implies that this approach does not face the generation of test data. Furthermore, M. Bernardino *et al.* [16] state that model-based testing "is a technique that takes advantage of the application model to generate testing models suitable for the generation of test artifacts", where *test artifacts* are appreciated to be test scenarios and scripts nor test data. Consequently, even if **Pre-1** holds, a test engineer is not able to derive synthetic workload with the aid of MBT only. This statement is enhanced by the depicted application of PLeTsPerf (cf. Figure 1 'Model-Based Performance Testing Process' in [16]) and Canopus (cf. Figure 3 'Model-based performance testing process using Canopus' in [5]), where the utilized data during test is marked as external resource each time. Thus for the MBT-based approach the **KQ-1** is to be negated.

Nevertheless, tools/IDEs like TestOptimal [18] exists, which

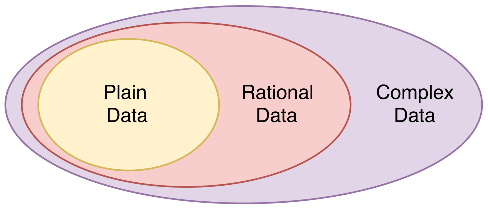


Figure 2: Illustration of chosen categories as well as the subset-relation between Plain, Relational and Complex Data Category

unite (test) data generation with the MBT-approach, a test engineer herself has to define data-columns or patterns to be used. Moreover, this data is only used as runtime data and does not solve the problem of missing base data, as required by **Pre-2**. Similar to the CR-based approach (cf. Section 4.2) this one itself is not capable of providing/compile workload nor test data as well. Thus to execute a performance testing, an external data source needs to be provided beforehand (answering **KQ-2**).

For an interim conclusion related to the research question, one can say that the retainment of test data / synthetic workload is not (yet) a part of one of the mentioned approaches. More precisely the data source is mostly treated as already available, external, resource, whereas its required to focus this remit more closely.

4.3 Approach Independent Opportunities

To provide an external data resource, required to adopt the previous mentioned approaches, we introduce various tools which accomplish this and moreover reveal possible, partial, solutions to the given test scenario (cf. Section 4.1).

In the field of (test) data generation many tools exists, for what reason it is suitable to decrease the extent of all tools. Since most tools are similar in a given manner, it is sufficient to group those in suitable equivalence classes and explain only one representative of each in detail. To achieve this, three different equivalence classes (hereafter 'categories') *Plain*, *Rational* and *Complex Data* (cf. Figure 2) facing the variations regarding resulting data complexity (cf. Section 4.3.1) are used. Based on this categorization representative tools are presented and exemplary used to generate data for *Bidder*.

4.3.1 Test Data Classification

Figure 2 depicts the three chosen categories. It illustrates that a subset relation $PlainData \subset RationalData \subset ComplexData$ between the categories exists. This correlation is applicable since every tool which can generate complex data is indeed capable of generating plain data as well.

The most inner category *Plain Data* represents all tools which are able to generate non-related and unordered data, e.g. single users-entries composed of name, surname and birthday. More comprehensive data generators which are able to create relational data fulfilling database constraints, e.g. foreign-keys, satisfying a relation between user entities and products (cf. Figure 1), are represented by the category *Rational Data*. Since a relation between different data entries can be omitted, this category is a superset of *Plain Data* as depicted in Figure 2. Finally, the category *Complex Data* enfolds all tools capable of generating complex data, i.e. data satisfying different application level constrains like ordered relations. E.g. for Bidder this is required, such that a sequence of bid's $b_1, b_2, ..b_n$ with different prices $p(b_i)$ could be generated. To obtain synthetic data all bid's need to be ordered as follows: $p(b_i) < p(b_{i+1})$. Similar to the previous category these additional characteristics could be omitted as well, whereas *Complex Data* tools are likewise able to generate *Rational Data* and furthermore *Plain Data*. Thus the subset relation between the chosen equivalence classes, $PlainData \subset RationalData \subset ComplexData$, holds.

4.3.2 Tools approach

Continuing on the data categorization, representative tools for each category are presented and exemplary adopted to the test-scenario. As it is apparent that the *Complex Data* category is best suitable for the given test-scenario, it will be presented in most detail.

Plain Data.

Generating plain data is nowadays a very simple task. Many tools, offline or online, exists to fulfill this task, e.g. faker.js [2], mockaroo.com [15] or generatedata.com [10]. Due to a (mostly) very intuitive user interface or good API/Framework (cf. Listing 1) it is simple to generate plain data, e.g. a set of user data. Thus generating a set of 100.000 unique users, include email and password, is very straightforward.

```
var faker = require('faker');
var users = [];
for( var i=0; i<100000; i++) {
  var user = {
    id: faker.random.uuid(),
    name: faker.internet.email(),
    password: faker.internet.password()
  };
  users.push(user);
}
```

Listing 1: Generating a set of 100.000 unique user, using faker.js

Relational Data.

Similar to *Plain Data*, the generation of relational data is very straightforward as well. Most tools analysis the metadata from an existing database (or alike) model and generate abstract entities based on this, likewise benerator (opensource) [3] and redgate sql [11]. Both tools use an existing sql-schema to deduce data models for generation. Beside this, when using benerator, one can describe the data using a lucid XML file. Listing 2 present a partial XML definition for benerator to generate relational data. Each generation of type *user* generates five related entities of type *product* holding the id of the user.

```
<generate type="user" count="100000">
  <variable name="person" generator="PersonGenerator"/>
  <id name="id" strategy="increment" />
  <attribute name="email" script="person.email" />
  <attribute name="password" type="string"
           pattern="[a-zA-Z0-9,.-_]{8,12}" />
  <generate type="product" count="5">
    <attribute name="user_id" script="{user.id}" />
    <attribute name="name" type="string"
             pattern="[A-Z][A-Z]{5,12}" />
  </generate>
</generate>
```

Listing 2: Partial XML exposing the generation of products related to a unique user using benerator.

It is obvious that the test data generation providing different products related to a generated user is achievable with benerator.

Complex Data.

The field of generating a huge set of complex data, e.g. ordered data, remains the most elaborating part. For example, and as required in the introduced test-scenario, a product can have arbitrary many Bids, which are sorted by means of *price* and *date*. Furthermore, to obtain plausible synthetic data, every bid has to be higher in price as the previous one (strict order). Consequently a appropriate tool has to allow rule or constraints definition, e.g. *state-based* generation, since the previous generated bid serves as basis for the next one.

For example *DataGenerator* by Finra is one tool which allows data generation by the usage of *SCXML* [6], defined by the Apache Common Foundation. Using DataGenerator the required data could be generator by a self-repeating state, guarded by a counter condition. Additionally, a custom *DataTransformer* is required to satisfy the constraints between different bids, i.e. each next bid has to have a higher price as the previous.

Beside DataGenerator the previous introduced *benerator* is likewise capable of generating *state-based* data by offering nested generation, which was already applied in for the *Relational Data* (cf. Listing 2) example. To provide a valid sequence of bids, varying in price, a *StepSequence* can be used, which generate a sequence using a fixed increment delta. But, even if the bids are distributed at random over all available user (excluding the owner, as shown in Listing 3, by variable *product*), the generated prices would only differ in a constant factor (defined by increment delta), which is not that realistic. For obtaining even more synthetic data the *RandomWalkSequence* is applied. It supports a `min` and `max` value (range) to vary the addition on each generation individually. But, it should be considered that this generated data, if applied multiple times to different systems, is not deterministic and thus could cause untraceable failures during test on the one hand. On the other its prevents developers from focusing/optimizing on concrete test data. Nevertheless, to provide synthetic data, the *RandomWalkSequence* is applied for the given test scenario, as exposed in Listing 3.

```
<generate type="bid"  consumer="database">

    <variable  name="product" type="product"
            source="database" distribution="random" />

    <generate type="bid_loop" count="25">
        <variable name="user" type="user" source="database"
                distribution="random" />

        <varibale name="price_ite" type="float"
                unique="true" distribution="randomWalk"
                minStep="0.5"  maxStep="10" inital="1" />

        <attribute name="user_id" script="{user.id}" />
        <attribute name="product_id" script="{product.id}" />
        <attribute name="price" type="float"
                 nullable="false" source="price_ite" />
    </generate>

</generate>
```

Listing 3: Partial XML exposing the generation of 25 ordered bids for each product using benerator.

To conclude the synthetic data derivation for the introduced test scenario one can say that various tools with different characteristics and features exists.

5. DISCUSSION

According to the previous findings, data generation and especially obtaining synthetic data is quite a difficult task, although various tools for different data complexity categories exists. We assume that this is due to missing knowledge and awareness of these tasks. Furthermore, these tasks are not covered as part of different approaches on performance testing as deducted with usage of **KQ-1** and **KQ-2** and summarized in Table 1.

	KQ-1	**KQ-2**
CR-based	No	requires external data resource
MBT-based	No	requires external data resource

Table 1: Compacted results regarding the key-questions for each approach

Both performance testing techniques are not capable of data deduction (**KQ-1**) and moreover treat synthetic workload as a given (external) resource which required during or afterwards the test artifact generations (**KQ-2**). This implies, that if these resources are not available prior to performance testing, additional work on artifact generation (test-data) is required.

To finally answer the motivated research questions (**RQ**), it is to say that data generation is not yet handled as part of performance testing. This is supported by the (currently) existing work and evaluations on this topic, as presented in Section 2. Thus a performance engineer is committed to take external or separated workload generation into account.

To support these results, i.e. excluding resource (test data) generation, one could argue that the fundamental definition of performance testing is (only) to evaluated and analyze a SUT under a certain load. Because these test are mostly done after a first roll-out, not in the early stages of development, enough data (runtime data), which only needs to be anonymized, exists and is not needed to generate nor to address as particular task. Thus the aspect of easy (automatic) load generation (scenarios and scripts) come to the fore and slights the aspects of data derivation. A possible minor improvement would be to compose workload generation tools and performance test asset deduction into a software-suite. Thus the tasks on synthetic workload generation and performance testing are kept separated, as argued by M. Utting *et al.* [19].

However, through the increasing development pace of applications, it gets more interesting to perform such test in early stages as well. As a consequence synthetic workload generation gains more priority, because no (runtime data) yet exists. On that account it would be a great advancement if both approaches (CR-based, MBT-based) would address and integrate the preparation of synthetic workload. Extending the CR-based approach towards thees enhancements is unfortunately inapplicable. This results, since the approach primarily relay on manual user (tester) input and does not hold any further information about the SUT, i.e. data model, which – in our means – prevents a (tool supported) synthetic data derivation. In opposition to that, the MBT-based approach could be extended to this means, since the test-scenario itself is available in a certain modeling language. A possible attempt could be to develop an intermediate language (DSL or similar) to combine existing tools and approach, e.g. *benerator* and PLeTsPerf. For instance a meta-model could be added to PLeTsPerf models to describe required data for a specific scenario. Based on this a compiler/translator, which is to develop, could be used to derive the require data model in a benerator convenient format. Consequently, a full performance test (single scenario) could be derived with the usage of one, specific, model only.

6. CONCLUSION

In this paper we have discussed tow different approaches, CR-based and MBT-based (cf. Section 3), towards performance testing and outlined in detail, utilizing **KQ-1** and **KQ-2** (cf. Section 1), to which extend these are capable of synthetic workload generation. We have deducted, with assistance of the two presumption **Pre-1** and **Pre-2** (cf. Section 4.2), that both approaches are not (yet) capable of data generation itself and show same results regarding the exploitation of test-data.

To provide these missing (synthetic workload) resources for performance testing we have introduced various tools – categorized by data complexity (cf. Figure 2) – to accomplish this (cf. Section 4.3). It was shown that, depending on the required workload complexity, the generation of this resource gets more complex itself (cf. Section 4.3). Exemplary we have introduced and utilized *benerator* to generate synthetic workload for the theoretically SUT *Bidder*.

Summarizing it is to say, that the current state-of-the-art regarding *synthetic data generation* and *performance testing* is not yet very contiguous. Even if the latter one is addicted to synthetic workload, both are mostly treated as different, disjoint disciplines. As a result the adaption of performance testing, without possessing test data prior, gets even more difficult. To decrease the amount of additional workload for a performance test in early stages of development, we have outlined and discussed two possible solutions. First the creation of more extensive test suits and second of all the development of an intermediate language to connect the different techniques, e.g. benerator and PLeTsPerf.

It is to outline that contemporary the research focus is set to easier and fast generation of test artifacts (cf. Test Generation). Prospectively the (automated) generation of test data, simultaneously with test -scripts and -scenarios, yields a proper research field and would result in very comprehensive tool-support during (performance-) testing. An already identifiable and likely possible solution towards these enhancements is, to create or extend existing model-based approaches with additional meta-models describing the required data for the specific scenario-based. This would allow an automatically deduction of test data (cf. Section 5).

7. REFERENCES

[1] F. Abbors, T. Ahmad, D. Truscan, and I. Porres. Model-based performance testing in the cloud using the mbpet tool. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 423–424. ACM, 2013.

[2] M. Bergman and M. Squires. faker.js - generate massive amounts of fake data in the browser and node.js. `https://github.com/Marak/faker.js`, 2016. [Online; accessed 2016-12-14].

[3] V. Bergmann. Databene Benerator. `http://databene.org/databene-benerator`, 2012. [Online; accessed 2016-11-02].

[4] M. Bernardino, E. M. Rodrigues, and A. F. Zorzo. Performance testing modeling: an empirical evaluation of dsl and uml-based approaches. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1660–1665. ACM, 2016.

[5] M. Bernardino, A. F. Zorzo, and E. M. Rodrigues. Canopus: A domain-specific language for modeling performance testing.

[6] A. Commons. SCXML - Commons SCXML. `https://commons.apache.org/proper/commons-scxml/`, 2015. [Online; accessed 2016-12-14].

[7] L. T. Costa, R. M. Czekster, F. M. de Oliveira, E. d. M. Rodrigues, M. B. da Silveira, and A. F. Zorzo. Generating performance test scripts and scenarios based on abstract intermediate models. In *SEKE*, pages 112–117, 2012.

[8] A. S. Foundation. Apache JMeter. `http://jmeter.apache.org/`, 1999. [Online; accessed 2016-11-02].

[9] IBM. Rational Performance Tester. `https://www-03.ibm.com/software/products/en/performance`, 2016. [Online; accessed 2016-11-02].

[10] B. Keen. generatedata.com. `https://github.com/benkeen/generatedata`, 2016. [Online; accessed 2016-12-14].

[11] B. Keen. SQL Data Generator - Generate realistic test data fast. `https://www.red-gate.com/products/sql-development/sql-data-generator/`, 2016. [Online; accessed 2016-12-14].

[12] H. P. E. D. LP. Load Testing Software: Application Testing Tools. `http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/index.html`, 2016. [Online; accessed 2016-11-02].

[13] Microsoft. Visual Studio IDE.

[14] Microsoft. Microsoft API and reference catalog - Chapter: Testing Performance and Stress Using Visual Studio Web Performance and Load Tests. `https://msdn.microsoft.com/en-us/library/ms123401.aspx`, 2016. [Online; accessed 2016-12-14].

[15] L. Mockaroo. mockaroo - Random Data Generator. `https://www.mockaroo.com/`, 2016. [Online; accessed 2016-12-14].

[16] E. Rodrigues, M. Bernardino, L. Costa, A. Zorzo, and F. Oliveira. Pletsperf - a model-based performance testing tool. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, April 2015.

[17] E. M. Rodrigues, R. S. Saad, F. M. Oliveira, L. T. Costa, M. Bernardino, and A. F. Zorzo. Evaluating capture and replay and model-based performance testing tools: an empirical comparison. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 9. ACM, 2014.

[18] L. TestOptimal. TestOptimal - Model-Based Test Automation. `http://testoptimal.com/TestOptimalFeatures.html`, 2015. [Online; accessed 2016-11-26].

[19] M. Utting and B. Legeard. *Practical model-based testing: a tools approach.* Morgan Kaufmann, 2010.

Black-Box Testing in the Presence of Database Inputs

Patrick Barakat
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
patrick.barakat@rwth-aachen.de

Konrad Foegen
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
konrad.foegen@swc.rwth-aachen.de

ABSTRACT

This paper introduces a combinatorial black-box testing approach for database driven applications. State-of-the-art approaches in combinatorial test design frequently focus on the applications input parameters or the database configuration. But they consider in general not the dependency of both. This paper describes how existing approaches could be used to describe a combinatorial method to generate a set of test cases out of a manual specified set of application, database input values and queries. Regarding the database state this leads to a better understanding of the behavior of the system under test (SuT) during the execution of test cases.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

combinatorial test design, test case reduction, black box testing, test heuristic, test case generation, data-intensive applications

1. INTRODUCTION

There exist large data-intensive applications today with hundreds of input possibilities and database tables. In this paper, we lay the focus on black-box tests of database-driven applications.

The test reduction is an important issue in software testing. Non-trivial software applications have an extremely large number of black-box tests [5]. One difficulty is to avoid the usage of the whole test set but still retain the fault detection capability [9].

In this approach, we distinguish between the application input values or short input values and database input values. Valid combinations (in refer to the database schema definition) of the database values are used for database operations such as insert, delete and update. The idea of using database inputs is especially then interesting, when there occurs database columns that could be not contained in the input values, e.g. this could be internal data such as an *order paid* flag for an online shop. In this case, there is an additional benefit through the database entry specification. The other way around there is an additional benefit when the manually specified input data contains parameters/values that are not contained in the database table. In example the application could perform a computation with some input value and store only the result in the database.

The next section gives an overview over several combinatorial black-box testing techniques. In section 3 an introducing example is provided. This example is based on the scenario of an order process to demonstrate the idea behind the afore mentioned heuristic. The feasibility of the approach is explained in chapter 4. The first part of the section 4.2 explains the general proceeding or idea behind our approach, whereas one possible realization of the approach is described theoretically in the rest of the section. The realization is based on the existing approaches of [1], [4] and[9]. Possible problems are mentioned in section 5. The paper ends with a conclusion in section 6.

2. COMBINATORIAL BLACK-BOX TESTING TECHNIQUES

In exhaustive test generation, there may occur test cases with combinations of input values that do not make sense or are even not allowed as input, like the combination of the day 31 and the month June. Therefore, reduction of test cases is an important issue in black-box testing. The techniques are often based on the source code of the application under test, the requirement speciation or large automation [4]. The reduction of test cases may be also done manually by the tester [7]. In practice the testers are not well supported in the search for "important" input combinations to test [9].

On the one hand, there exist theoretical approaches like t-way testing and especially pairwise testing [10] with a predefined interaction strength $t = 2$ between the input parameters. On the other hand, there are practical applications of this approach. Among these are [9] and [11].

Although pairwise testing is widely used it is in general not recommended [2]. But there exist many approaches that enhance this strategy by using additional heuristics like [1] and [9].

These approaches have the objective to reduce the tests by additional information on the interaction between input and output [9] or the (t-way) testing as in example described in [1].

Alsewari et al. describes an approach is to reduce the test cases in presence of constraints [1]. The constraints are defined in this approach as a combination of input values that is not allowed to occur. The method uses the Late Acceptance Hill Climbing based Strategy (LAHC), i.e. it is an adaption of the LAHC algorithm.

The LAHC algorithm is a heuristic optimization algorithm. LAHC starts with an arbitrary solution for a problem and tries with local search in every iteration to find a better solution by changing just one element of the solution [3]. The algorithm terminates when no better solution can be found.

Alsewari et. al. [1] adapts the algorithm by using an optimization function to maximize weight of the covered interactions. The actual LAHC-based algorithm starts with an initialization. After that the test cases are computed iteravely by using a diversification and an intensification strategy. The algorithm ends when all interactions are covered. During the diversification, a solution, or more precise in this case an uncovered pair of user specified sample values is picked. In the same iteration, the intensification is done by using a probabilistic perturbation function to find an optimal (local) solution. This function uses a heuristic to exploit the search space by local or global search. The perturbation function decides by probabilities whether the solution should be modified and which direction (in the neighbourhood) for the search should be choosen. The found solution will be added to the test suite if there is no constraint violated.

There are several other strategies that are based on reduction of test cases with help of an interaction relationship, for example [11] and [6]. But according to Alsewari et al. the LAHC algorithm seems to be the first algorithm that uses the LAHC method to reduce test cases [1].

Schroeder and Korel introduced an approach to reduce the number of possible black-box tests by automated input-output analysis [9]. The aim is to identify relationships between program inputs and outputs to reduce the number of combinatorial tests. The idea behind that is to omit test cases with input values that don't have influence on the applications' output. This leads to a reduction of combinatorial test cases without losing the fault detection capability. It is also not necessary to have multiple test cases that leads to the same program output. Thus, the idea is to choose a minimal set of combinations of input values, such that every output is covered once.

For the selection of input data, black-box test data selection criteria (such as equivalence-class partitioning, boundary value analysis, etc.) can be applied [9]. The relationship between the program input and output could be determined manually, e. g. by analyzing the program documentation. Schroeder and Korel suggests also techniques, like the structural analysis of source code. However, we determine the input-output relationship also automatically but in a different way than proposed by [9]. This is explained in more detail in section 4.

3. MOTIVATING EXAMPLE

In this section, we show an introducing example of the proposed heuristic. The SuT will be a fictive online shop application with a SQL database schema. In this example, we will just consider an excerpt of the system, namely an input mask for submitting the order and the corresponding database tables. One part of the input of the algorithm is the structure of two database tables: the table order and the table client. All database tables are supposed to be already available in the SuT. Furthermore, the tester should provide a set of input parameters with possible input values. The input parameters consist of application input parameters and database input parameters. In this paper we distinguish between input parameters, which refer to the type of variable or a specific database table column such as a client no. and the input values that describe the different characteristics of a parameter (e. g. the client no. $1, 2, \ldots, n$). The input parameters, as well as the input values are manually defined by the user. Beside that the tester may also specify constraints which restricts the occurrence of specific input combinations. Suitable inputs for the database tables and the input masks are then provided by the algorithm. This is explained in more detail in section 4. The structure of the tables 2 and 1 is from the SuT, whereas the data in the table is generated by the algorithm. To achieve a deterministic test results, we suppose to have a predefined database state before inserting data sets. In this example, we assume that we have no productivity data in the database.

Table 1: Table client

Client no.	Shipping address (*address*)	Payment method (*payment*)	Reminder procedure
1	Angle alley 7	Paypal	No
2	Downing street 100	Payment in advance	Yes
3	*NULL*	*NULL*	No

Table 2: Table order

Order no.	Order value	Client id
5	500	1
10	105	2

Orders can be stored in table 2. They are simplified given by an order no., an order value and a corresponding client no. Table 3 shows some manually defined client data (a client number, a shipping address, and a flag, whether the client is in a payment method procedure). The user or respectively tester might get these sample values by methods like boundary-value analysis or equivalence class partitioning.

In the following we provide a set of predifined database queries. These should be formulated in way that they all may lead to successful running test cases. Thus they should not select invalid combinations of data.

1. Client that is registered
2. Client that has a payment method
3. Client that has no reminder procedure
4. Clients with shipping address and payment method that are not new and not empty

To 1: Only registered clients should have the opportunity to order.

To 2: A valid payment method is necessary for the order.

Table 3: Input and configuration data

Client no.	Shipping address	Payment method	Query
1	Angle alley 7	Paypal	Client has a payment method
2	King's Road 1	Payment in advance	Client with no reminder procedure
3	Downing street 100	Debit	Shipping address and payment method are not new
4	*NULL*	*NULL*	Client is registered

Table 4: Database configurations

Query	Input data dependence	Allowed input values	Disallowed input values
1	{client no.}	$\{1, 2, 3\}$	$\{4\}$
2	{client no., payment}	$\{1, 2\}$	$\{3, 4\}$
3	{client no.}	$\{1, 3\}$	$\{2, 4\}$
4	{client no., address, payment}	{(1, Angle alley 7, Paypal), (2, Downing street 100, Payment in advance)}	{t| t=(x,y,z), x=1,..,3, t *not* in query 4}

Table 5: Generated test cases with pairwise testing

Test no.	Client no.	Shipping address	Payment method
1.	1	Angle alley 7	Paypal
2.	1	King's Road 1	Payment in advance
3.	1	Downing street 100	Debit
4.	1	*NULL*	*NULL*
5.	2	Angle alley 7	*NULL*
6.	2	King's Road 1	Paypal
7.	2	Downing street 100	Payment in advance
8.	2	*NULL*	Debit
9.	3	Angle alley 7	Debit
10.	3	King's Road 1	*NULL*
11.	3	Downing street 100	Paypal
12.	3	*NULL*	Payment in advance
13.	4	Angle alley 7	Payment in advance
14.	4	King's Road 1	Debit
15.	4	Downing street 100	*NULL*
16.	4	*NULL*	Paypal

Table 6: Generated test cases with database heuristic

Test no.	Client no.	Shipping address	Payment method
1.	1	Angle alley 7	Paypal
2.	1	King's Road 1	Payment in advance
3.	4	King's Road 1	Debit

To 3: It is not possible to order when there is a reminder procedure for the client.

To 4: Should increase the safety level for the client. The payment method should be verified if the address changes and there should be also a verification after the payment method changes.

Table 4 shows the input data dependencies from the sample values in table 3 for every query. The idea behind that table is to clearify the dependency to the valid and invalid input values projected on the corresponding query information.

Table 6 provides the selected test cases by the combinatorial algorithm. This selection is one possible solution and depends on the concrete implementation of the algorithm mentioned in 4.2. The goal here is to generate a minimal set of test cases for allowed input values such that every query is considered. For possible test cases that don't belong to set mentioned before, e.g. they have valid input values that should lead to a failure, we can act analogously. Another idea might be to select the test cases by the expected behaviour, e.g. the expected output values. In this case, it might be more reasonable to select the test cases such that the sets are most disjoint.

Here test no. 1 is selected because of the 4th query. He covers with client 1 every other defined query. Test no. 2 is selected because of the disallowed values for query 4, whereas test no. 3 is selected because of the disallowed values for queries 1 - 3. We suppose that it is not possible to place an order with the values specified by the test cases 2. and 3. All in all the selection should (simplified) be done by selecting input data dependencies such that every query is covered. This will be explained in more detail in section 4.2.

4. COMBINATORIAL TESTING IN PRESENCE OF DATABASE INPUTS

In the following subsection 4.1 a coverring array (CA) and of a mixed covering array (MCA) structure is explained. They could be used in order to quantify or describe how well a set of test cases covers all combinations of the corresponding sample values. In addition, we explain an extension which considers the CA and MCA together with constraints. The idea of the combinatorial testing in presence of database inputs is explained in detail in subsection 4.2.

4.1 Specification of input data

We assume to have an empty database. This has the advantage that no faults could be misdiagnosed due to inconsistend data entries. The database is given by a set of tables $D = \{T_1, \ldots, T_n\}$ with $n \in \mathbb{N}$. Where the table T_i with x_i columns, y_i rows and the set of data entries $\{d_{iab} | 1 \leq a \leq y_i \text{ and } 1 \leq b \leq x_i\}$ is given by

$$T_i = \{(d_{i11}, \ldots, d_{i1x_i}), \ldots, (d_{iy_i1}, \ldots, d_{iy_ix_i})\}, \text{ for all } 1 \leq i \leq n$$

We formalize a result of a query as a projection π on a specific index set $I \subseteq \{1, \ldots, x_{i_1} + \cdots + x_{i_k}\}$ of the tuple set

$$T_{i_1} \times \ldots \times T_{i_k}, i_1, \ldots, i_k \in \{1, ..., n\}, k \leq n$$

The queries should be projected only to columns that corresponds to the set of database input parameters. Other parameters are not relevant for the algorithm presented in 4.2. Thus, they can be omitted.

We define a test case as a tuple $(t_1, \ldots, t_l)$, where l is the number of input parameters according to the test case. Each input parameter has a space of possible values $t_i \in P_i \subseteq \Sigma^*$, where Σ is an arbitrary machine readable alphabet. Without loss of generality we can assume that P_i is a finite set. In the case that

$$|P_1| = \ldots = |P_l|$$

the test coverage can be described by a Coverring Array $CA(N, t, v^p)$ (CA) structure. It is a $N \times p$ matrix or respectively array, where all t-way interactions are included, each row describes a test and every column a parameter [8]. Every possible t-tuple is supposed to be contained at least once in every $N \times t$ subarray. The parameters p, v, and t in the coverring array structure describe the number of parameters, values (or levels) and the interaction strength [10].

The idea behind the interaction strength is to to detect faults that are caused by interactions between any t parameters. Thus, the interaction strength t can be considered as the maximum number of parameters that is assumed to interact with each other. In example in pairwise testing there is an interaction strength of 2. The t-way testing could be considered as generalization of the pairwise testing, where arbitrary interaction strengths t are possible. An exhaustive test could be considered as a t-way test, where t equals the number of parameters.

Let us consider the covering array $CA(6, 7, 2^2)$ as an example. This CA has 6x7 arrays with an interaction strength of 2, whereas the exhaustive test would lead to $CA(128, 7, 2^7)$ or in other words 122 test cases more than the pairwise test.

The CA can only be used to describe the number of test cases with an interaction strength for paramaters that all have the same number of possible values, as mentioned above. To specify the test coverage for multiple paramaters with different characteristics a mixed covering array $MCA(N, t, C)$ could be used [10]. The parameter C is defined by $v_1^{p_1} v_2^{p_2} \cdots v_n^{p_n}$ indicating that there are p_1 parameters with v_1 values, p_2 parameters with v_2 values and so on.

Furthermore we define the constraints covering array (CCA) or mixed-constraints covering array (MCCA) as defined in [1], where a new set F of forbidden interactions is introduced (i.e., $(CA(N, t, v^p, F)$ or $MCCA(N, t, C, F))$. According to [1] the set F contains tuples with elements of the format $F_{i,j}$ where i indicates the i-th parameter p or respectively p_i and j indicates the j-th value or respectively v_j of the parameter p_i. In example the set $F = \{(F_{i,j}, F_{k,l})\}$ expresses that the j-th value of the i-th parameter is not allowed to occur together with the l-th value of the k-th parameter.

4.2 Feasibility of the approach

As mentioned in section 1 we have a set of queries Q, a set of constraints C and a table consisting of possible values for the application input parameters, as well as for the database input parameters that are used as input of the algorithm. The definition of the sample values for the application input data and the queries of the algorithm input could look like in table 3. We have assumed to have an (initial) empty set of constraints in the example in section 3. For comprehensibility we started with a possible resulting database tables (compare table 2 and 1). The constraints could be in the form of a $MCCA$ that prohibits different combinations. This idea could be expanded to the whole input of database and application values. However, it is more useful to consider the database parameters and input parameters separately instead of the whole input. In this case, it might be possible to define different interaction strengths. Nevertheless, we supposed to have additionally a set of constraints C_{AD} between both input parameter types. In example this is useful when the application input parameter and database parameter are related in the sense that an input in the application could lead to change, e. g. an insert or update of the corresponding value of the database parameter. Assume that we have the sets T_A and T_D that satisfies the corresponding constraints. The goal is to generate a combinatorial set of test cases that are reduced by considering the interaction between the database and application input values. One reduction can be achieved with the help of the queries and constraints. This can theoretically be done by first considering the set of constraints C_{AD} between T_A and T_D. Then we have a (maximal) set $T_{AD} \subseteq T_A \times T_D$ of possible test cases that satisfies the constraints, i. e. it is not possible to add another combination without violating a constraint. This set could be further reduced by incorporating the query information Q. It is possible to compute a relation between input parameters and database parameters M or in other words which query has an influence on application input values or vice versa. This leads to the idea to select test cases T of T_{AD} such that every query is tested and further reduction would result in queries that are not covered. From another perspective adding more test cases should result in redundant tests.

With the specified input, we can reach our goal in three different steps:

1. Derivation of combinations between the sample values of the application and the sample values of the database, under the condition that the result contains only valid combinations in refer to the constraints.
2. Generate database content.
3. Specify dependencies between input values and queries and select test cases such that all queries are covered by the relation at least once.

There are several ways to realize the reduction of test cases in presence of database constraints. To achieve a comparable result to the algorithm we use different algorithms that were already mentioned in section 2 like the LAHC based algorithm of [1] and an adaption of the input-output analysis method from [9]. Furthermore, we use an algorithm or more precise a method that has similarities to the application of the approach of Chays et al. [4].

As already mentioned the tester describes sample values of certain attributes/parameters for the application input (I_A) and for the database tables (I_D). The goal is to create test cases from the set I_A and input data sets from I_D. As explained in 2 there exist many approaches to generate black-box test cases from input values. The generation of database content from sample values is more involving because the content has to fit to the database schema. In example, foreign key and not null constraints has to be satisfied.

For each query, it is possible to define dependencies to the application input values. In example an address field of a database table and the corresponding address field on the input mask of the application. These dependencies could in general not be resolved automatically due to the different or ambiguous naming of table attributes and input parameters. When we consider the sets I_A and I_D with different name spaces, the introduction of an optional alias for every parameter could solve the problem. In this case, it is possible to compute the dependencies automatically by naming the alias of a parameter p that is used in I_A just as the alias of a corresponding parameter p' from I_D.

With this dependency information it is possible to compute the input data dependency as shown in table 4 and then retrieve the possible input values that were specified manually by solving/executing the corresponding query. It is sufficient to get the values that fit to the query by solving the query conditions. Each result is stored in a relation M between the power set of the database parameters and the set of the application input values. Thus according to the example in section 3 it holds $(\{\text{client no.}\}, \{1, 2, 3\}) \in M$.

That followed a set of constraints can be computed from the information in M. For each $(D, P) \in M$ and every pair of elements in D or respectively P it is possible to compute constraints for combinations which are not allowed in refer to the application input values. These constraints are added to the input set of constraints C_A. The idea is to define this relationship by using the queries. Let $Q_1, \ldots, Q_k$ be the result sets for each query $q_1, \ldots, q_k$. With this information, it is possible to derive constraints. For $(d_1, d_2, \ldots, d_{im}) \in Q_i$ we deduce a constraint set F such that the input elements have to map on the corresponding elements of the database. More precisely for some d_j we take the disallowed values in refer to the corresponding input parameter j as constraints.

Beside the definition of constraints for the application input values it is useful to define also constraints for the database values and attributes manually. Thereby it is possible to avoid invalid input combinations.

The generated set of test cases may furthermore be reduced by an input-output analysis as described in [9]. As mentioned earlier the queries should be specified in a way that all of their executions lead either to a valid or an invalid result. The idea is now to use the input-output analysis method to reduce the merged test cases. The goal of this step is to reduce the test cases to a minimal set in the sense that every test case covers at least one query and a reduction of one test case would break that condition. This can be achieved by storing an additional flag that contains the corresponding queries to the disallowed inputs 4.

The mentioned approach has the disadvantage that it is not comfortable to handle foreign key constraints. All the database columns have to be specified and with that it is possible to set up which combinations are not allowed. These might be a large set of constraints for every key / foreign key combination, namely all values that don't equal each other.

Another way to achieve the goals mentioned in the beginning of this section is the use of a similar application as AGENDA [4].

AGENDA provides components to fill a database with data and to generate inputs to the application under test [4]. Furthermore, it is possible to specify values that should be treated differently by the SuT in so called data groups.

The set of database constraints C_D is partitioned into the sets $\tilde{C}_D$ and $\bar{C}_D$. The difference between $\bar{C}_D$ and $\tilde{C}_D$ is that the set $\bar{C}_D$ defines constraints between different database attributes (e. g. foreign key constraints) and has also as a subset the relation between different values of a certain attribute. These values define the before mentioned data groups and are expected to be threatened differently by the application. The set $\tilde{C}_D$ describes combinations that are not allowed as input, in example a shipping date that is before the date of the order. The idea behind the data grouping is similar to the data grouping of the AGENDA algorithm [4]. The grouping is supposed to define a relation between different sample values of a specific attribute. A group can be defined when the database sample values are treated differently by the SuT.

Listing 1: Algorithm in pseudocode

```
1   Input:  Query set $\mathbf{Q}$ ,  database schema $\mathbf{D}$ ,
2           $I \subseteq I_A \times I_D$ ,
3           optional set of constraints  $C = C_A \dot\cup C_D \dot\cup C_{AD}$
4   Output:  Set of test cases $\mathbf{T}$  and database queries $\tilde{D}_I$
5   $M \leftarrow$ computeDependencies$(Q, C_A, C_D)$;
6   $\bar{C}_A \leftarrow$ computeConstraints$(M)$;
7   $C_A \leftarrow \bar{C}_A \cup C_A$;
8   $T_A \leftarrow$ LAHC$(I_A, C_A)$;
9   $T_D \leftarrow$ LAHC$(I_D, \tilde{C}_D)$;
10  $\tilde{D}_I \leftarrow$ generateQueries$(T_D, Q, \bar{C}_D)$;
11  $T' \leftarrow$ LAHC$(T_A \times T_D, C_{AD})$;
12  $T \leftarrow$ IOA$(T', M)$;
```

As mentioned earlier the decision criteria to select the test cases as in [9] focus on the input combinations that affects the programs output. But here we need just to consider the relation M in a way such that all queries are covered by the input data dependencies (compare table 4).

An algorithm in pseudocode is shown in 1. As already mentioned the algorithm has the input Q, D and a (manually) defined set of sample values I. The method *computeDependencies* assigns the dependencies to M. The idea behind that is indicated in table 4 of the example in section 3. With the dependency relation it is possible to state the set of constraints $\bar{C}_A$ which is added to the input set of constraints C_A. After that possible test cases for the application are generated by the LAHC-method with the I_A and the expanded set of constraints. Analogously possible combinations of database values that satisfies the constraints C_D are generated by the LAHC-method. The purpose of $\tilde{D}_I$ is to generate the SQL commands for the test cases, e. g. to fill the database with data. The method *generateQueries* is supposed to generate a query for every combination of data groups. In line 11 the generated application and database inputs are combined with each other by the help of the constraints in C_{AD}. This could be considered as a merging step. Because the input of the set C_{AD} is optional, there may occur combinations of application and database input values that cannot exist in practice. Therefore, an input-output analysis is performed by the IOA-method at the end to achieve a further reduction of test cases according to the relation M. The result of this method and the set $\tilde{D}_I$ are the output of the algorithm.

5. PROBLEMS

As mentioned in section 4.1 it is requested to specify input values as well as database values and queries for the input of the algorithm. In industrial applications, it is not uncommon that they consist of large table sets. The table sets may also have dozens of columns. It could be a tedious work to specify all constraints manually. The manual specification of the queries could be also a hard work. It makes sense to gather them from the application, e. g. an application tracing might provide the executed sql commands.

The algorithm considers only the database structure, i.e. it does not consider the current database state. This may make it hard to combine the tests with non-combinatorial tests. The algorithm could be slightly modified such that the table data is at first not generated by the *LAHC*-method or that also predefined database entries are considered by the combinatoric. Furthermore, it could lead to problems to apply the approach of [1] directly to compute database combinations because input combinations are in general not denied for all queries.

The realization of the approach let some important things about the usage out: the question is in which order the input values and the database values should be inserted. There are use cases for both opportunities: As described in the example it may be useful to have “predefined” database entries. The other case is that the database entries are considered as a result of the corresponding input.

AGENDA generates queries for both cases. Let us consider an example similar to the Unit Replay Model in [8]: in addition to the described input values there may also occur buttons like “order now” and “save order”. One can assume that the corresponding buttons lead to different datasets of the order table, e.g. they have a different order status. Then the expected behavior of the application depends on which button was pressed and therefore also the test cases that could be applied. In example, it should not be possible to click “save order” for placed order because it should be already saved.

6. CONCLUSIONS

In this paper, we have shown an approach that uses beside the usual application input values also database inputs and queries in order to reduce the set of combinatorial test cases. The consideration of the database configuration leads to the advantage that data-intensive applications could be tested more efficiently and also more effectively than without knowing information about the database state.

The idea behind the approach of [9] seems to be easy. Nevertheless, the realization of such an approach could be for data-intensive applications a tedious work. This has the reason that in general not all input/output relationships could be analyzed automatically. Additionally, the preconditions for the input variables cannot be retrieved automatically in general. But in this approach these disadvantages play no role because the information could be gathered automatically from the input.

In this approach, we also introduced an algorithm that uses the LAHC-method. On the one hand this approach is quite new and practical experience is missing. On the other hand, the authors claim to have promising results and are still working on an improvement of the method [1].

7. REFERENCES

[1] A. A. Alsewari, K. Z. Zamli, and B. AL-Kazemi. Generating t-way test suite in the presence of constraints. *Journal of Engineering and Technology (JET)*, 6(2):52–66, 2015.

[2] J. Bach and P. J. Schroeder. Pairwise testing: A best practice that isn’t. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*, pages 180–196, 2004.

[3] E. K. Burke and Y. Bykov. The late acceptance hill-climbing heuristic. *University of Stirling, Tech. Rep*, 2012.

[4] D. Chays, J. Shahid, and P. G. Frankl. Query-based test generation for database applications. In L. Giakoumakis, editor, *Proceedings of the 1st international workshop on Testing database systems*, page 1, New York, NY, 2008. ACM.

[5] C. Cheng, A. Dumitrescu, and P. Schroeder. Generating small combinatorial test suites to cover input-output relationships. In H. Lin, editor, *Proceedings / Third International Conference on Quality Software, QSIC 2003*, pages 76–82, Los Alamitos, Calif., 2003. IEEE Computer Society.

[6] G. Demiroz. Cost-aware combinatorial interaction testing (doctoral symposium). In M. Young and T. Xie, editors, *2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 440–443, New York, 2015. ACM.

[7] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006.

[8] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. Chapman & Hall / CRC innovations in software engineering and software development. CRC Press, Boca Raton, FL, 2013.

[9] P. J. Schroeder and B. Korel. Black-box test reduction using input-output analysis. In D. J. Richardson, editor, *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 173–177, New York, NY, 2000. ACM.

[10] K. Z. Zamli, A. A. Alsewari, and M. I. Younis. T-way testing strategies. In I. Ghani, W. M. N. W. Kadir, and M. N. Mohammad, editors, *Handbook of research on emerging advancements and technologies in software engineering*, Advances in systems analysis, software engineering, and high performance computing (ASASEHPC) book series, pages 421–433. Engineering Science Reference/IGI Global, Hershey, Pa., 2014.

[11] W. Ziyuan, N. Changhai, and X. Baowen. Generating combinatorial test suite for interaction relationship. In M. Pezzè, editor, *Fourth international workshop on Software quality assurance in conjunction with the 6th ESECFSE joint meeting*, page 55, New York, NY, 2007. ACM.

Continuous Delivery for Enterprise Architecture Maintenance

Peter Hansen
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
peter.hansen@rwth-aachen.de

Simon Hacks
RWTH Aachen University
Software Construction
Ahornstr. 55
52074 Aachen, Germany
simon.hacks@swc.rwth-aachen.de

ABSTRACT

Currently enterprise architecture management is mostly a manual activity that requires ongoing maintenance. In recent years, techniques to automatically collect data for the enterprise architecture evolved. However, there are challenges with those: There might be conflicting changes and when data is collected from different sources, they might not share the same semantics. In this paper, we will examine how practices from continuous delivery, which is typically used in the context of software engineering, can help to cope with those problems.

Categories and Subject Descriptors

K.6.1 [**Management of computing and information systems**]: Project and People Management—*Strategic information systems planning*

Keywords

enterprise architecture (EA) management, continuous delivery, automated EA documentation

1. INTRODUCTION

Enterprise architecture currently is mostly modeled manually and changes require huge manual efforts. This is especially true when complex organizational structures need to be covered and the organization is constantly changing. The pace of changing structures and complexity is expected to increase and this makes it even more challenging [19]. In recent years, the field of enterprise architecture management already adopted techniques to reduce model maintenance effort. However, there are still challenges in regards to conflicting changes, different semantics and responsibilities [5].

In the field of software engineering, changing requirements are also very common. Software engineering deals with this by becoming as agile as possible and uses various social and technical techniques to improve towards this direction [15]. Examples for social techniques are the ongoing adoption of agile process models like scrum or kanban and even techniques directly related to the development itself like pair programming. Technical examples are the rise of continuous integration and delivery. All of these techniques lead to the same shared goal: Shorten feedback loops [11]. Techniques used for software engineering are also being adopted for other parts of organizations: With the DevOps movement, which emphasis on the collaboration of development and operations, infrastructure is being covered using techniques typically used in the context of software engineering and processes are also adopted [3].

In this paper, we introduce the important aspects of enterprise architecture management and continuous delivery. Then, we compare the relation between those concepts and highlight shared goals. Based on those, we create an example pipeline for enterprise architecture maintenance and discuss how the enterprise architecture goals are reflected within this pipeline. Later, we discuss the advantages and disadvantages of this example pipeline and conclude to which extend the challenges regarding enterprise architecture can be solved by the application of continuous delivery.

2. EA TERMINOLOGY

According to Bernard, enterprise architecture (EA) is "the analysis and documentation of an enterprise in its current and future states from an integrated strategy, business, and technology perspective" [1]. Commonly, this information is represented within a model and reports are generated from the model to provide decision support to various kinds of stakeholders.

Thus, enterprise architecture management (EAM) describes the processes around developing and maintaining the EA. This involves information gathering from various sources including manual collection as well as structured information from external systems. The enterprise architecture does also reflect future states, thus a part of EAM deals with developing those and supports the transitions [8]. In summary, EAM deals with the following tasks:

- document the current state
- develop a target state
- support the transition from current state to target state

It is hard to fulfill this as enterprises are constantly changing to meet the current and future needs of their customers to stay competitive. These changes also affect business processes as well as supporting information systems. Enterprise architecture management is a central discipline in this changing world as it should provide information and support the changes. As the information is only relevant when the model is up-to-date, the maintenance is an essential part of EAM.

2.1 Data Quality

As EAM has a huge effect on organizations, the data quality of the EA model is of particular interest. Farwick et al. used a survey to research on what are the most important data quality attributes. The results of this survey were (ranked by importance, most important first) [5]:

Correct Granularity (EA01)
Data has a meaningful level of abstraction.

Consistency (EA02)
Data should not contain errors or contradictions.

Actuality (EA03)
Data should be up-to-date.

Completeness (EA04)
Data covers all elements of relevance.

As the participants of the survey were also allowed to propose own attributes, the survey also yield two additional attributes without qualitative ranking on importance:

Ownership (EA05) Who is responsible for the data

Traceability (EA06) What is the origin of the data

As - at least for the last two attributes - we did not find any other survey that evaluates the importance of those attributes. We have also seen that further work of Farwick et al. focuses on data actuality [7], granularity [6] and a combination of consistency and actuality [4]. We see this as indication that all of these properties are relevant.

2.2 Federated EAM

EA is used in large organizations and information that is used within the EA is often owned by different departments. This makes it hard for a central enterprise architecture team to gather all information and keep them up-to-date. Fischer et al. proposed a federated approach for the maintenance of EA models [8]. The main idea is that the data is kept within specialized architectures and linked to a central EA repository. Because this is an organizational as well as technical challenge, the authors propose a maintenance process that involves different roles. We refer to this federated property of EA as **EA07**. All process activities and all involved roles with their responsibilities are shown in Figure 1. We will briefly describe the activities and roles that are used within this process and start with the roles:

EA coordinator Business oriented role; manages the maintenance process and focus on specification of interfaces and reports.

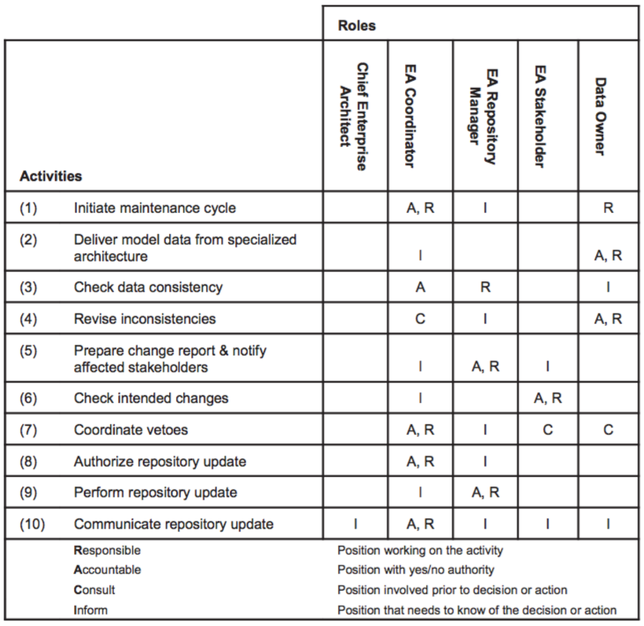
EA repository manager Technical oriented role; is responsible for technical operations of the EA repository.

EA stakeholder the contact persons in the units that are using the EA and providing input.

Data owner Owners of specialized architectures.

The **chief enterprise architect** is not directly involved in the process but is informed on updates.

From the activity perspective, the process starts with an initiation activity (1) where either a data owner or the *EA coordinator* starts the process. The next activity, delivery specialized architectures (2), requests all *data owners* to provide their model data. After that, a consistency check (3) is performed where the provided model data is checked. The revision of inconsistencies (4) is only required if the consistency check (3) failed for a certain specialized architecture. In the change report and notification activity (5), a report is created that compares current data of the EA with the new data set. This report is distributed among the *EA stakeholders* asking them for feedback on the changes. The *stakeholders* then check if the changes are intended (6). In case of problems, there is an activity to resolve conflicts (7). If there are no problems or the conflicts are resolved, the repository update is authorized by the *EA coordinator* (8). Then, the update is executed (9) and everyone involved is notified about the update (10).



		Roles				
Activities		Chief Enterprise Architect	EA Coordinator	EA Repository Manager	EA Stakeholder	Data Owner
(1)	Initiate maintenance cycle		A, R	I		R
(2)	Deliver model data from specialized architecture		I			A, R
(3)	Check data consistency		A	R		I
(4)	Revise inconsistencies		C	I		A, R
(5)	Prepare change report & notify affected stakeholders		I	A, R	I	
(6)	Check intended changes		I		A, R	
(7)	Coordinate vetoes		A, R	I	C	C
(8)	Authorize repository update		A, R	I		
(9)	Perform repository update		I	A, R		
(10)	Communicate repository update	I	A, R	I	I	I
	Responsible	Position working on the activity				
	Accountable	Position with yes/no authority				
	Consult	Position involved prior to decision or action				
	Inform	Position that needs to know of the decision or action				

Figure 1: Activities and responsibilities of the EA maintenance process [8]

2.3 Automated EA

We have seen that the enterprise architecture maintenance is a process that contains individual activities. Towards automation the first question that arises is when to start the process. Fischer et al. proposed that the maintenance activity should be executed in regular intervals but it might also be required to execute it manually when things change [8]. Farwick et al. identified that there are change events that could be used to trigger the EA maintenance process [7]. The paper presented several event sources that could be used to generate those events. We reference to the existence of such change events as **EA08**.

One source that was used as an example by Farwick is using the enterprise service bus (ESB) to gather information. An ESB is a message bus that is used throughout an organization to integrate distinct applications. This technique was later used by Buschle et al. [2]. Another approach was the usage of network scanners to identify services and infrastructure [12]. Despite that, many other systems could be used as event source and trigger model maintenance.

Regarding automatic data collection and model maintenance, Hauder et al. have researched on challenges in this context [10]. As a result, the main concern is the ”abstraction gap to EA model”. This means that the data from automatic collection does not fit to the EA model. We have seen an analogous result in the survey mentioned in Section 2.1, which states that correct granularity is important.

3. CONTINUOUS DELIVERY

Continuous delivery (CD) is a relatively new topic in the field of software engineering and was firstly coined by Humble et al. in 2006 [14]. It describes the practice to automate the build, test and deployment process leading to shortened release cycles. Continuous delivery is formulated using a *deployment pipeline* which describes the flow of an artifact through different build and test stages. In difference to the practice of continuous delivery, *continuous deployment* extends this concept by deploying every change that passed the pipeline automatically to the production environment [9]. In the following, we will highlight the main properties of continuous delivery (as described in [13, 14]).

3.1 Automation

Without automation, continuous delivery would not be possible: Build, testing and deployment tasks have to be repeated over and over again on each code change. But the automation has more implications than to reduce manual effort. In the past, deployment was typically a manual, complex and error prone task. The error proneness is not necessarily only due to human mistakes. One reason is that the way the software has to be deployed might change during development and the deployment documentation does not co-evolve. Another typical reason is that the infrastructure changes due to updates and does not correspond to the deployment documentation anymore. In a CD setup, the automation around the deployment is put under version control and is executed on a regular basis (on each code change). Thus, it is ensured that the deployment is always working and corresponds to the current software version and the current infrastructure (referenced as **CD01**). The documentation problem is also solved because a separate documentation of the deployment process is no longer necessary.

In current approaches, the deployment aspect has gained more attention and is more comprehensive: The aspect of the infrastructure is also described as code (”infrastructure as code” [17]) and thus behaves similar to regular application code in some aspects. When putting application development and system operations together, this is usually called DevOps [3].

3.2 Traceability

When building a project that depends on another library, the library and the project itself should not be build separately. Instead, a source code change in one of the repositories (library or project) should lead to a full rebuild of the project. Otherwise, source code changes in the library that change test results for the main project are hard to identify. We reference traceability as **CD02**.

3.3 Same artifacts for all environments

Artifacts that are created during the CD process should be independent of the target environment. Thus, there should be only one artifact and not one for the production environment and one for the testing environment. If configuration needs to be changed for different environments, a separate configuration repository should be specified and used during deployment.

When having this ”one artifact for all environments” property (**CD03**), it is always ensured that the version on production is exactly the same as the one that was tested before.

3.4 Pipeline

In continuous delivery, the deployment pipeline models the process from check-in to release. We reference this formal process description as **CD04**. The process consists of activities that are grouped in stages and also includes quality gates (referenced as **CD05**). Examples for those quality gates could be passing unit tests, a threshold of a metric that is checked or even the result of a manual test activity.

An example pipeline is shown in Figure 2. During development, changes are made in the source and configuration repository. After this, the software is compiled and unit tests are executed based on the source code repository. If this stage passes, the resulting artifact is uploaded to the package repository. The next stage, functional testing, takes the artifact from the previous stage (unit test) and loads configuration information from the configuration repository. Other test stages (QA, Performance, UAT) might work in the same way as functional tests but may require configuration information of a different environment. After all test stages are executed, the artifact from the package repository could be deployed to the production environment using a production configuration from the configuration repository. The use of the package repository ensures that exactly the same build of the artifact is used throughout the whole process. The tagging arrows shown in the figure are referring to version tagging in the version control system.

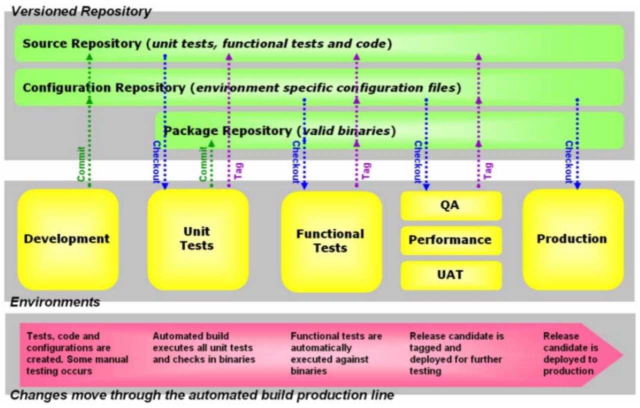

Versioned Repository
Source Repository (unit tests, functional tests and code)
Configuration Repository (environment specific configuration files)
Package Repository (valid binaries)
Commit
Checkout
Commit
Tag
Checkout
Tag
Checkout
Tag
Checkout
Development
Unit Tests
Functional Tests
QA
Performance
UAT
Production
Environments
Tests, code and configurations are created. Some manual testing occurs
Automated build executes all unit tests and checks in binaries
Functional tests are automatically executed against binaries
Release candidate is tagged and deployed for further testing
Release candidate is deployed to production
Changes move through the automated build production line

Figure 2: Continuous delivery pipeline [14]

4. IDENTIFYING SHARED GOALS

When looking into enterprise architecture and continuous delivery, we have identified a bunch of commonalities on their goals. We will look into the individual properties of continuous delivery and discuss how they relate to the goals of enterprise architecture.

4.1 Low risk releases

A key concept of continuous delivery is that it facilitates low risk releases. This is achieved by the following continuous delivery concepts:

- automated deployment of the software
- reproducible build that can be used across all environments
- defined pipeline that will be used for all builds

This usually leads to a more frequent release cycle. Which in turn leads to earlier feedback if things are broken or do not correspond to the requirements.

Putting these in the context of enterprise architecture, it is obvious that it does perfectly fit:

- The central enterprise architecture model, which is used for reports that support strategic business decisions or change projects, is comparable to a production environment and changes should be evaluated carefully before they are put into this model. However, when evaluating changes to the model in a different "environment", the change to the production "environment" reflecting the central model should be exactly the same as the one that was reviewed (reproducible build, defined deployment).
- The pipeline idea is also applicable: There are multiple steps that might be necessary to deliver a model: Merge the data, check consistency, check if quality gates are passed, manual review stages and finally trigger report generation. If this process would not be automated, it would be hard to ensure that each step is actually executed and no shortcuts are used by the people involved.
- The short release cycles are also a good fit as we have seen that it is crucial to have up to date reports to decide upon.

4.2 Lower costs

Continuous delivery reduces the manual effort to release a build in each environment. This goal also fits to enterprise architecture as cost efficiency is important for its acceptance in the organization.

4.3 Better products

Continuous delivery uses short release cycles to get user feedback earlier. When using this feedback, it leads to happier customers. As this is true for software, this is also true for enterprise architecture. The customers are the stakeholders of the enterprise architecture that depend upon reports. For the enterprise architecture managers, it is also good to receive early feedback as it might discover errors earlier and thus safe money.

5. EXAMPLE PIPELINE

In this section, we propose an example continuous delivery deployment pipeline for enterprise architecture that is based on the maintenance process by Fischer et al. [8] which we described in Section 2.2. In our approach, we use the term "data source" instead of specialized architecture to show that the input does not necessarily represent architectural data but could also be in the form of configuration databases (CMDB), license management tools or other databases. The roles are the same as used by Fischer et al., however, we combined activities to a shared stage and added automation proposals.

Commit stage is triggered by a change event (*EA08*, see Section 2.3) of a data source in the federation (*EA07*). The data is fetched from the data source and committed to the version control system. The data owner *EA05* needs to be identified in this step: If the data source delivers such information, the commit is executed in charge of the person who have changed the underlying data. If the data source does not support to deliver such information, the owner of the data source is assumed as the responsible person for the change. If the data source does not support to provide change events, we need to poll the data sources regularly. This means that it might be a problem to track a change down to one person. In this case, the best possible solution is to use the data owner for all changes. Because of polling or unfiltered change events, the following changes are only executed if data in the version control system is actually changed.

Lint check checks if the input data matches the expected format (e.g. valid xml, syntax check). This stage starts multiple jobs in parallel: For each input source one job is started. The stage passes if all jobs were successful. If any job fails, the whole stage fails and the pipeline execution is stopped immediately.

Normalization The data is normalized to a certain file format. This is also a stage containing parallel jobs for each data source. If any normalization fails, the stage will fail. A failed normalization might uncover semantical errors in the input data.

Integration The result of this stage is one integrated model that is derived from the input data. This stage will execute the model modifications in a defined order and abort if any modification fails. The imported data is annotated with the source of the data *EA06*. Possible reasons for failed modifications are conflicting data or missing integration points.

Consistency stage carries out a statical analysis of the model, e.g. by checking if relationships could be resolved *EA02*. If inconsistencies are found, the stage fails.

Acceptance Based on the created model, all reports are generated in exactly the same way as they are generated by the production system. If report generation fails, the stage fails. Otherwise, the reports could be reviewed. If the changes are correctly reflected within the reports, the stage completion is manually acknowledged by the reviewer.

Deployment Optionally, the model could be deployed to the production environment. As in the acceptance phase, all reports are generated.

When comparing this pipeline with the process shown in Section 2.2, we see that there is a mapping of the activities to the stages of the pipeline. Activities 1 and 2 map to the commit phase. 3 and 4 map to linting, normalization and integration while no. 5-8 are represented by consistency and acceptance. The final activities 9 and 10 are covered by the deployment stage.

As an addition to the process of Fischer et al., our proposal is extended by a stage for **automated acceptance testing** that is executed between consistency and manual acceptance test. We have seen that enterprise architecture is not only about capturing the current state, but also about change management. When information about the current state is collected automatically, it would also be good to check if changes are in accordance to the planned changes. An advantage of continuous delivery in this scenario is that it states to have full builds on each change. In EA this would mean if any underlying information source changes, the build would be triggered. Thus, deviations from the planned changes would be detected early by the enterprise architecture team.

An example scenario for this is the following: A company plans to introduce a new software system on all desktops. This software has defined hardware requirements but some of the existing desktops are too old to run the new system and thus should be replaced with newer hardware. The hardware specifications are automatically collected from a central inventory system. When change events are detected on the inventory system, the delivery pipeline is executed. A comparison of the old and new model shows that a new desktop was found. As the system was identified as new and the necessary specifications are defined, an automated test is able to check for compliance with the planned state.

6. DISCUSSION

In the previous section, we presented our pipeline proposal. In this section, we discuss how the pipeline relates to the properties for continuous delivery and enterprise architecture that we have identified in the beginning. Additionally, we discuss the advantages and disadvantages of using the proposed pipeline for real world enterprise architecture maintenance.

The properties of continuous delivery are reflected by the pipeline as follows:

Co-evolution of pipeline and software (CD01) is contained in the pipeline, because the release consists of a set of reports which is created in exactly the same way for the acceptance test environment as for the production environment. Thus, if the model should derive other reports in production, it must also do so in acceptance test stage. A further observation is that additional data sources could only be integrated in the process by changing the pipeline. Thus the co-evolution is designed in the pipeline approach.

Traceability (CD02) is realized using change events that are processed individually for each data source. One commit and thus one build therefore contains only changes from one source. Even if polling is used, there is still only one data source contained in the commit.

Same artifact for all environments (CD03) The merged model that is used to create the reports in the acceptance test phase is the same that will be released as production model. Continuous delivery requires the reports generated from the model for the acceptance test stage to be the same as the ones that are delivered for production changes.

Formal process (CD04) The process is described using activities that are grouped in stages. We have also defined what happens if things break and what should be executed in parallel. This information allows formal description of the process.

Quality gates (CD05) are build in: All lint, consistency, merge and report operations may fail due to invalid data. The acceptance test stage contains a manual approval gate.

Now we discuss the enterprise architecture properties:

Correct granularity (EA01) As the data is derived from external systems and we do not have filters implemented, the process does not ensure anything about granularity. We see two possibilities to overcome this issue: 1. The step that merges the model of a certain data source into the global model could implement data source specific filters. 2. As this seems to be a general problem with EA, it would be sensible to implement a filter in the tools that work on the model.

Consistency (EA02) The process contains a consistency check phase. However, semantical conflicts may not be discovered in this step. Example: If the inventory system contains a server with name "database" which is the same as "db" from a license management software, it is hard to map the properties of this server to the same model elements. We argue that the manual acceptance test should discover such defects and recover from it by introducing a common naming across all connected systems.

Actuality (EA03) is guaranteed by either using the change events or poll on the data sources. As the whole process runs on each change, the actuality of the production reports basically is limited in the ability to do manual acceptance testing on the reports.

Completeness (EA04) The reports and models are as complete as the data sources and the model allow. If parts of the organization are e.g. not covered by inventory systems, the information could not be (automatically) added to the model and finally this leads to an incomplete model. Nevertheless this is not a drawback of our pipeline but an organizational challenge of EA.

Ownership (EA05) We propose committers to be change owners and thus be responsible for the last version of the data. We introduced a fall back mechanism to ensure everything has an owner.

Traceability (EA06) is ensured by having small changes only from individual data sources. Additionally, the merging step adds the origin of the data to the model if possible.

Federation (EA07) The EA is federated as different data sources could be connected.

Existence of change events (EA08) The change events are used to trigger the process.

For practical implementation, we expect two problems:

- Small changes have high overhead when using events: Even small inventory changes trigger a whole build and require manual acceptance. In the prior-CD process, small changes would be aggregated due to common sense.

- We proposed to do commits per data source. If one piece of information is stored in multiple systems (e.g. the server in the license management and inventory), the changes are committed individually. As in the physical world the creation of a new server in several system is a kind of atomic change, this should also be reflected in the process.

Basically, these problems lead us to a scheduling problem: When should the pipeline be executed to aggregate a sensible amount of data, but keep the changes still small enough to be able to review the changes manually. Our opinion on this is, that a right balance between those can only be found by using the process and may differ across organizations.

7. CONCLUSION

In this paper, we provided an example continuous delivery pipeline proposal for enterprise architecture maintenance that is based on an existing maintenance process found in literature. As the continuous delivery approach allows to introduce new stages, we introduced the concept of automated acceptance testing for enterprise architecture changes. To the best of our knowledge, there is currently no research on this type of test. One evidence that we found is the work of Schneider et al. that derives entropy information from enterprise architectures [18]. However, this would not include any directional information towards the enterprise architecture target state. From our current understanding, this type of test does have practical relevance and could be a driver for automated enterprise architecture documentation.

Then, we discussed how this pipeline fits to the enterprise architecture domain as well as to the continuous delivery approach based on properties we found in literature. We discussed the drawbacks of the process.

As future work, the proposed pipeline needs to be implemented in a real world scenario and the actual impact on the enterprise should be evaluated, especially with focus on test description and execution. The work of Khosroshahi et al. already suggests that there are various organizational challenges when a decentralized EA model is introduced [16].

8. REFERENCES

[1] S. Bernard. *An Introduction to Enterprise Architecture: Third Edition.* AuthorHouse, 2012.

[2] M. Buschle, M. Ekstedt, S. Grunow, M. Hauder, F. Matthes, and S. Roth. Automating enterprise architecture documentation using an enterprise service bus. 2012.

[3] P. Debois. Agile infrastructure and operations: how infra-gile are you? *Agile, 2008. AGILE '08. Conference*, pages 202–207, 2009.

[4] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke. Automation processes for enterprise architecture management. In *2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops*, pages 340–349, Aug 2011.

[5] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke. Requirements for automated enterprise architecture model maintenance - a requirements analysis based on a literature review and an exploratory survey. In R. Zhang, J. Cordeiro, X. Li, Z. Zhang, and J. Zhang, editors, *ICEIS (4)*, pages 325–337. SciTePress, 2011.

[6] M. Farwick, W. Pasquazzo, R. Breu, C. M. Schweda, K. Voges, and I. Hanschke. A meta-model for automated enterprise architecture model maintenance. In *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*, pages 1–10. IEEE, 2012.

[7] M. Farwick, C. M. Schweda, R. Breu, K. Voges, and I. Hanschke. On enterprise architecture change events. In *Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation*, pages 129–145. Springer, 2012.

[8] R. Fischer, S. Aier, and R. Winter. A federated approach to enterprise architecture model maintenance. *Enterprise Modelling and Information Systems Architectures*, 2(2):14–22, 2007.

[9] M. Fowler. Continuous delivery. http://martinfowler.com/bliki/ContinuousDelivery.html, 05 2013. Accessed on 2016-12-18.

[10] M. Hauder, F. Matthes, and S. Roth. Challenges for automated enterprise architecture documentation. In *Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation*, pages 21–39. Springer, 2012.

[11] J. Highsmith and A. Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

[12] H. Holm, M. Buschle, R. Lagerström, and M. Ekstedt. Automatic data collection for enterprise architecture models. *Software & Systems Modeling*, 13(2):825–841, 2014.

[13] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[14] J. Humble, C. Read, and D. North. The deployment production line. In *Proceedings of the Conference on AGILE 2006*, AGILE '06, pages 113–118, Washington, DC, USA, 2006. IEEE Computer Society.

[15] J. Jeremiah. Survey: Is agile the new norm? https://techbeacon.com/survey-agile-new-norm, 2015. Accessed on 2016-12-18.

[16] P. A. Khosroshahi, S. Aier, M. Hauder, S. Roth, F. Matthes, and R. Winter. Success factors for federated enterprise architecture model management. In *Advanced Information Systems Engineering Workshops - CAiSE 2015 International Workshops, Stockholm, Sweden, June 8-9, 2015, Proceedings*, pages 413–425, 2015.

[17] K. Morris. *Infrastructure as Code.* O'Reilly Media, second early release (2015-11-02) edition, 2015.

[18] A. W. Schneider, T. Reschenhofer, A. Schütz, and F. Matthes. Empirical results for application landscape complexity. In *System Sciences (HICSS), 2015 48th Hawaii International Conference on*, pages 4079–4088. IEEE, 2015.

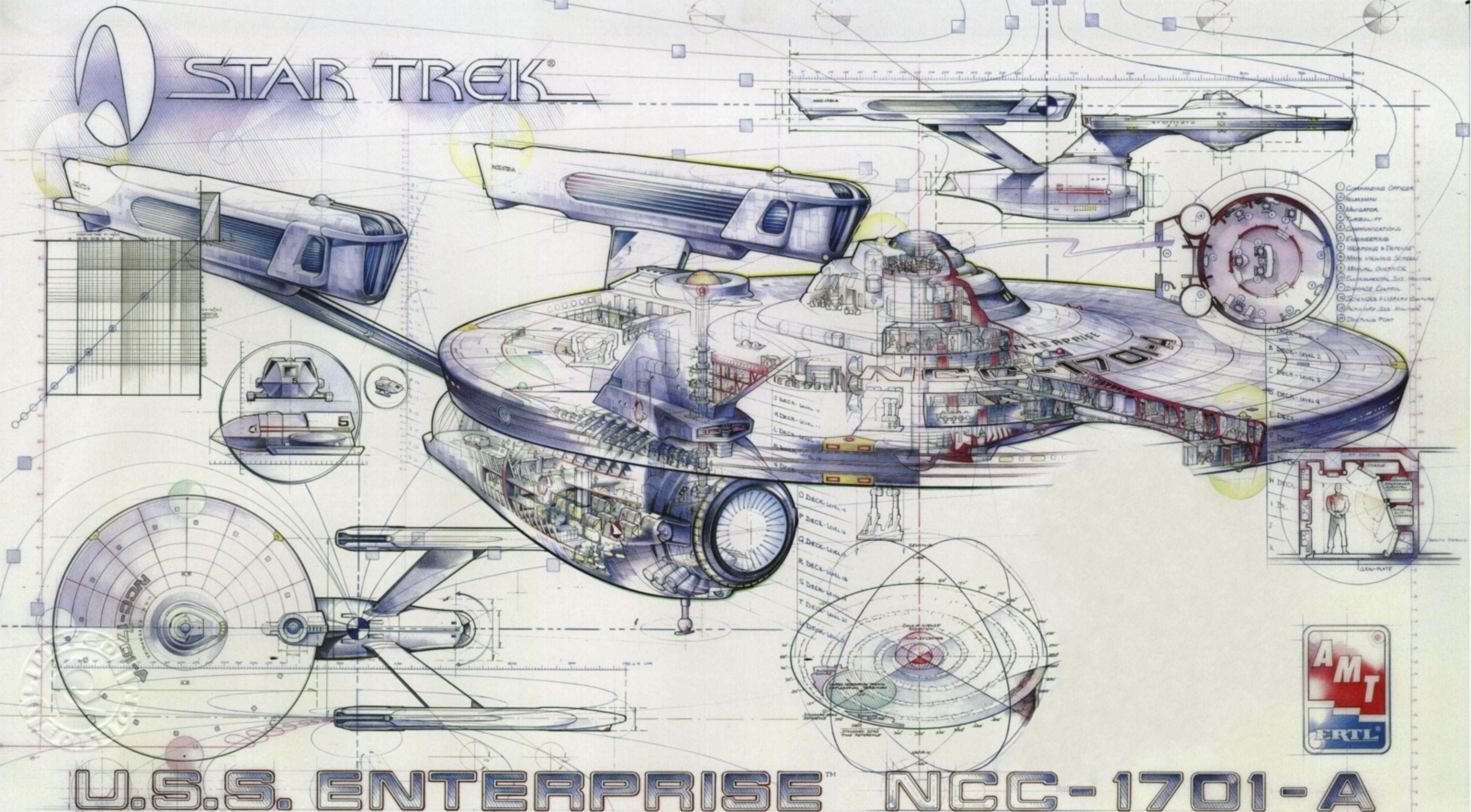
[19] S. M. F. Winter, Katharina; Buckl and C. M. Schweda. Investigating the state-of-the-art in enterprise architecture management methods in literature and practice. In *MCIS 2010 Proceedings. 90.*, 2010.

Peter Hansen, peter.hansen@rwth-aachen.de

Continuous Delivery for Enterprise Architecture Maintenance

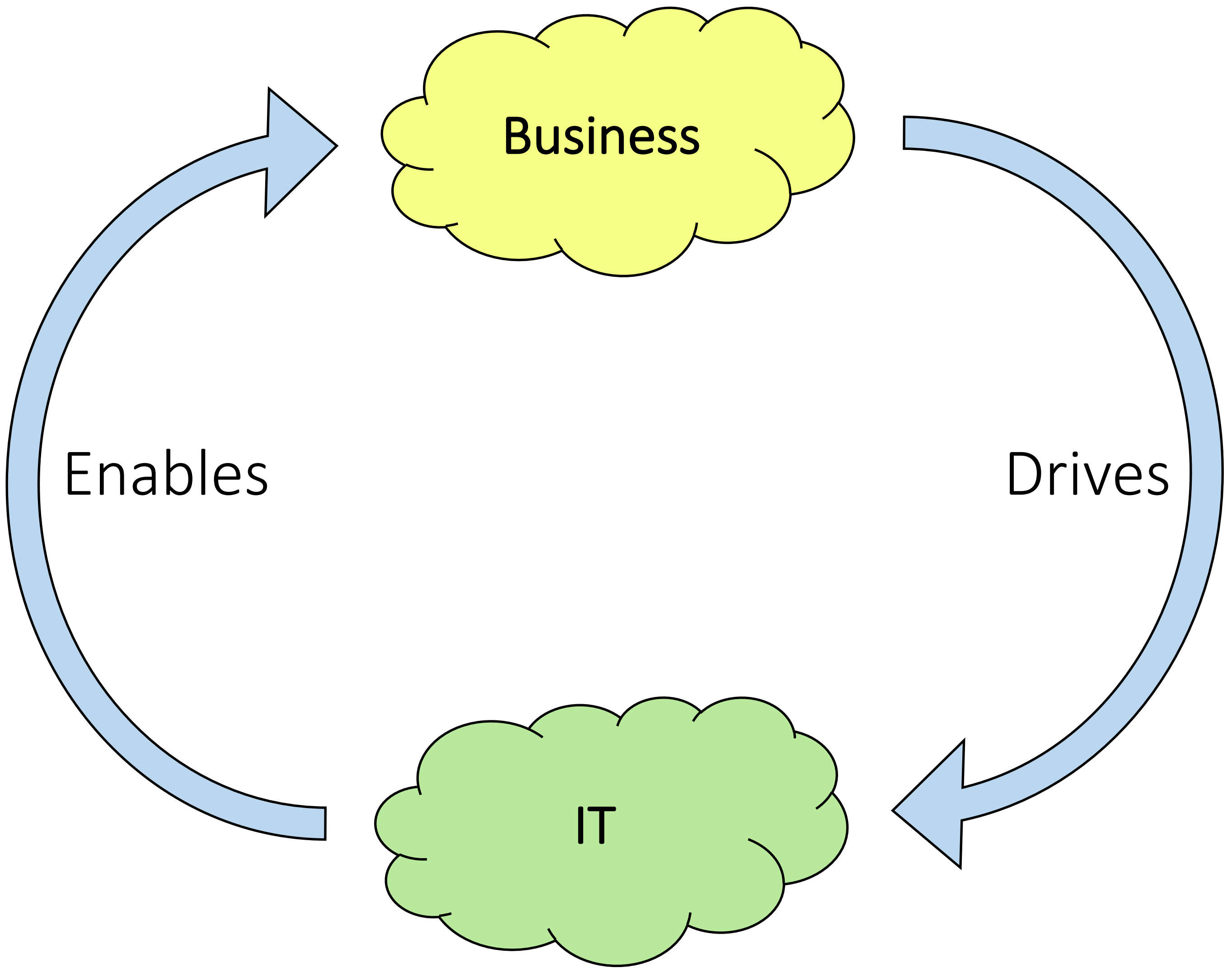
“If it hurts, do it more often!”

STAR TREK®

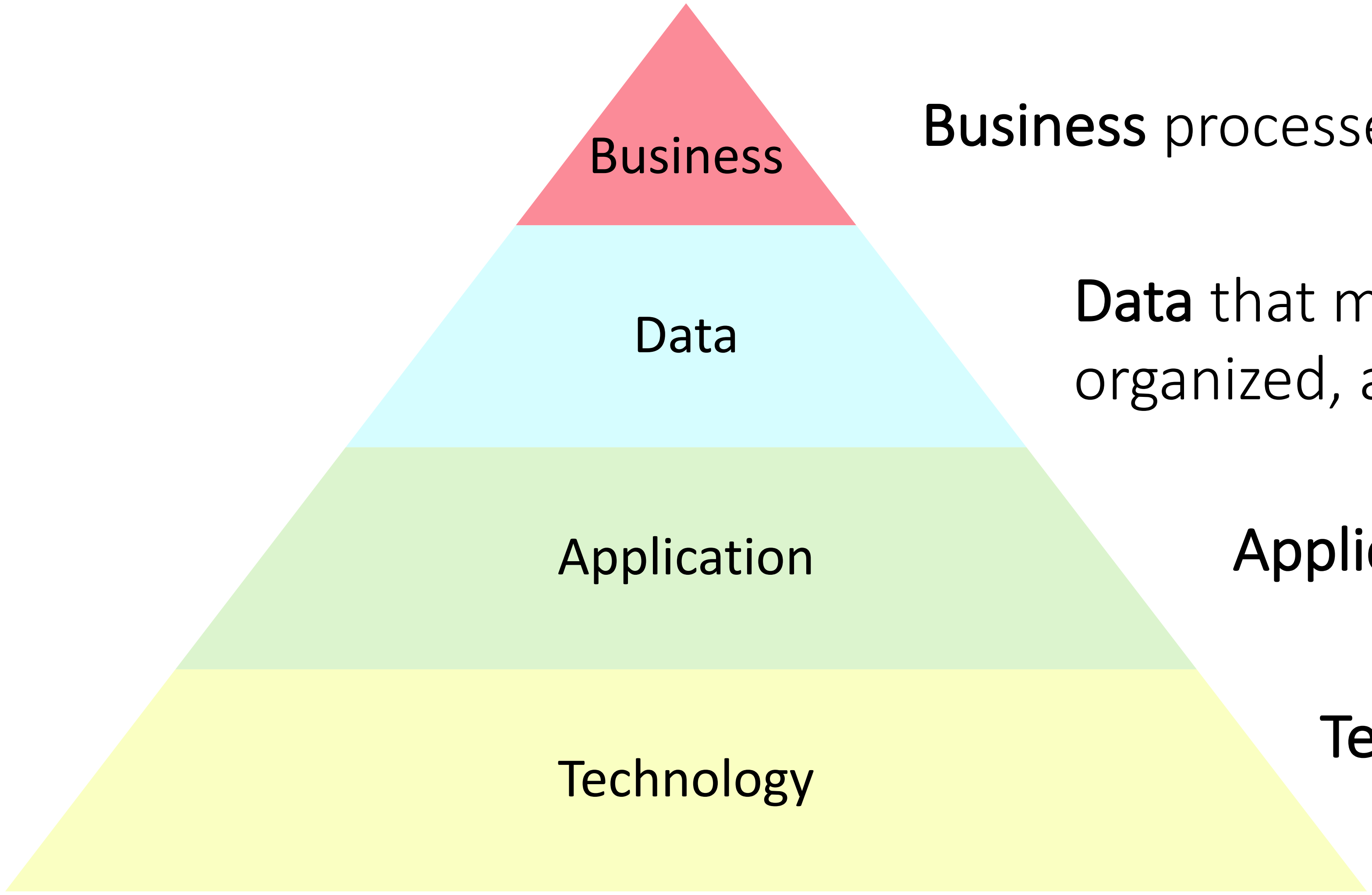


U.S.S. ENTERPRISE™ NCC-1701-A

Business IT Alignment



Enterprise Architecture Concerns



Business processes and activities use...

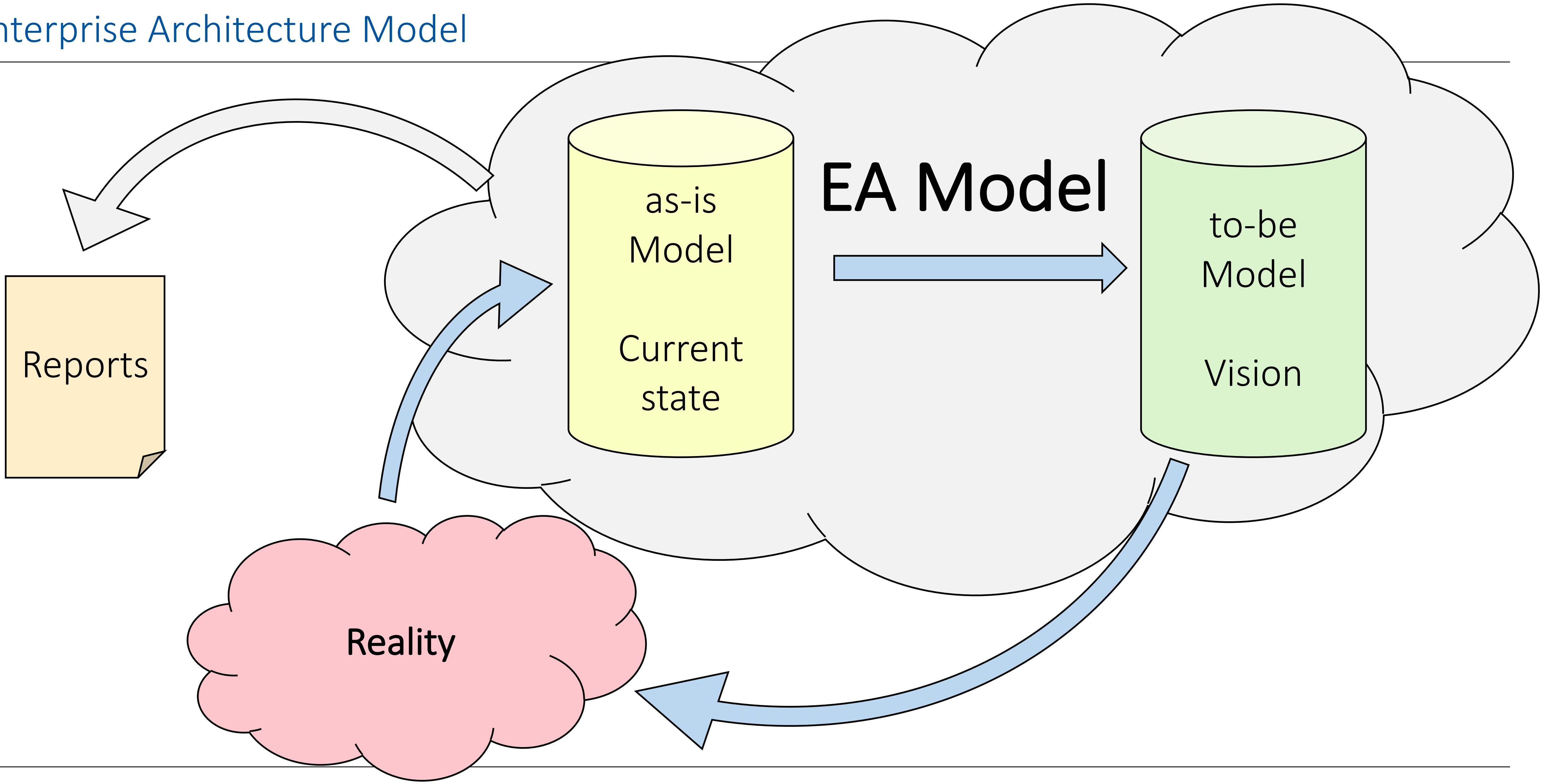
Data that must be collected, organized, and distributed using....

Applications that run on...

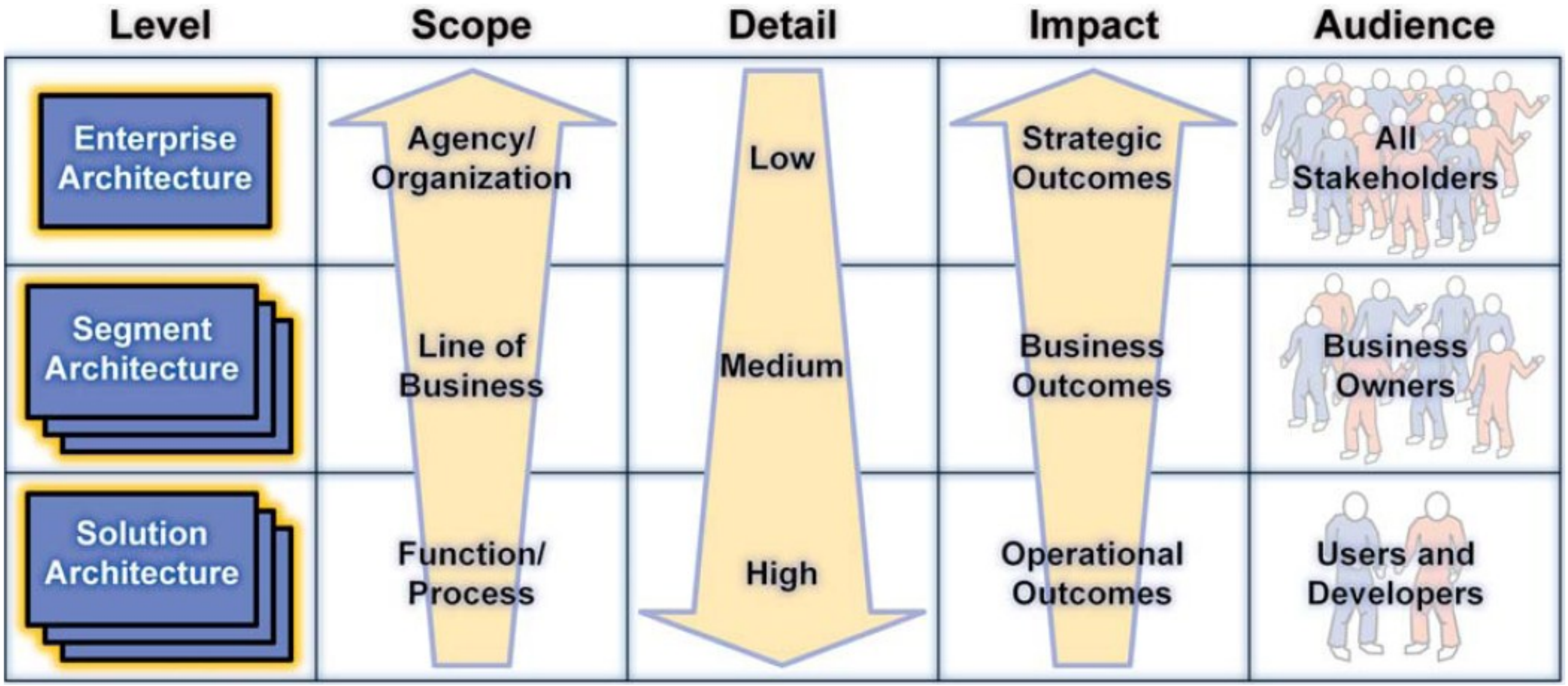
Technology such as computer system and networks.

Niles E Hewlett, 2006

Enterprise Architecture Model



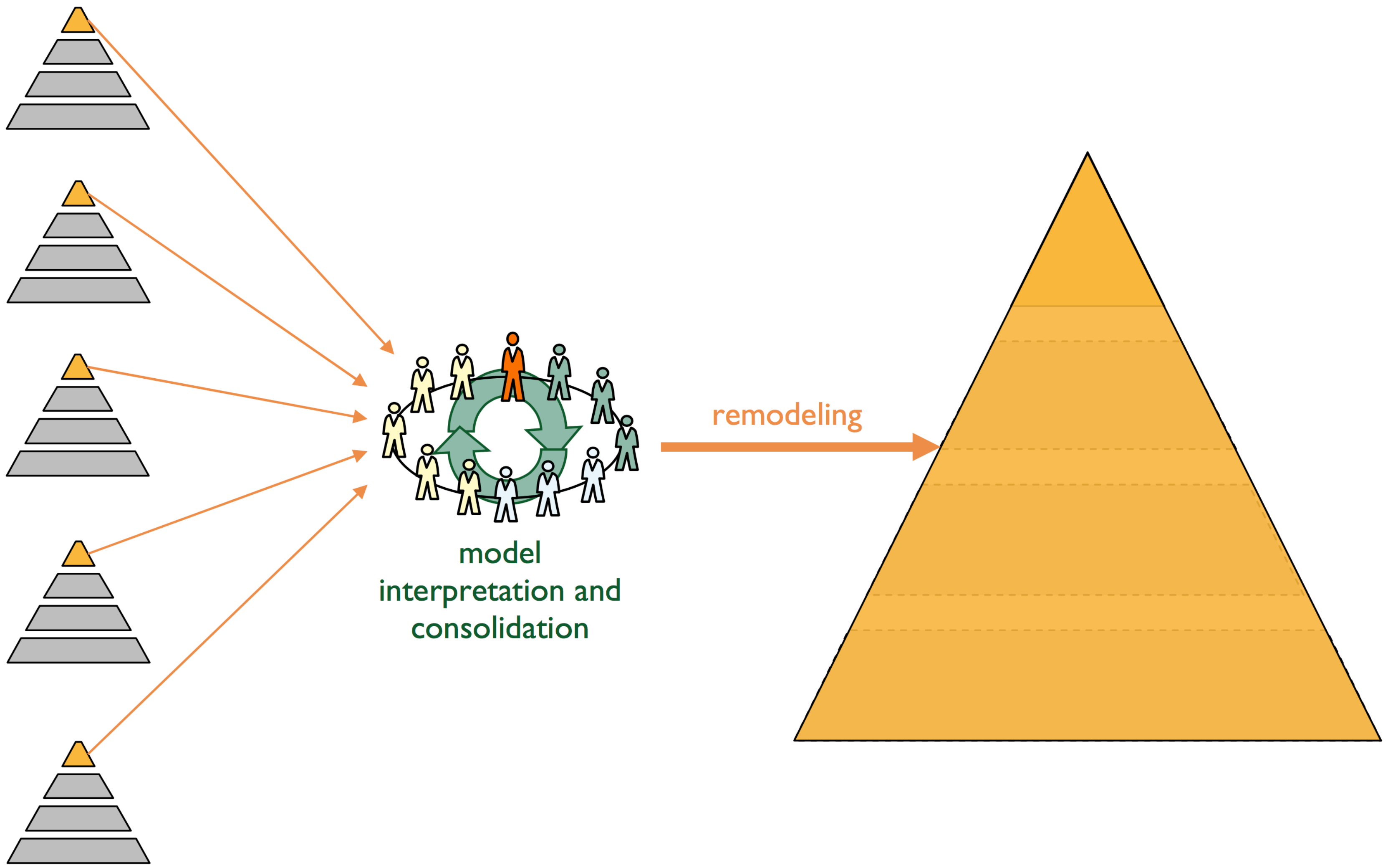
Enterprise Architecture Abstraction



Level	Scope	Detail	Impact	Audience
Enterprise Architecture	Agency/ Organization	Low	Strategic Outcomes	All Stakeholders
Segment Architecture	Line of Business	Medium	Business Outcomes	Business Owners
Solution Architecture	Function/ Process	High	Operational Outcomes	Users and Developers

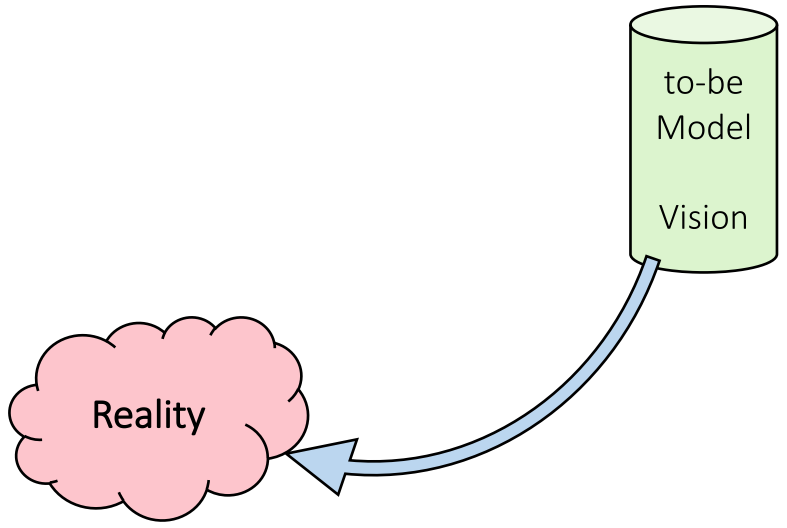
Federal Enterprise Architecture Program Management Office, 2008

Enterprise Architecture Federation

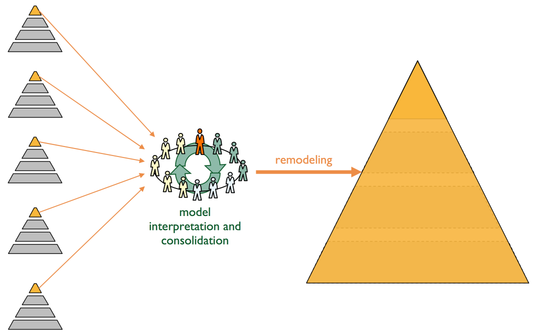


Stephan Aier, 2007

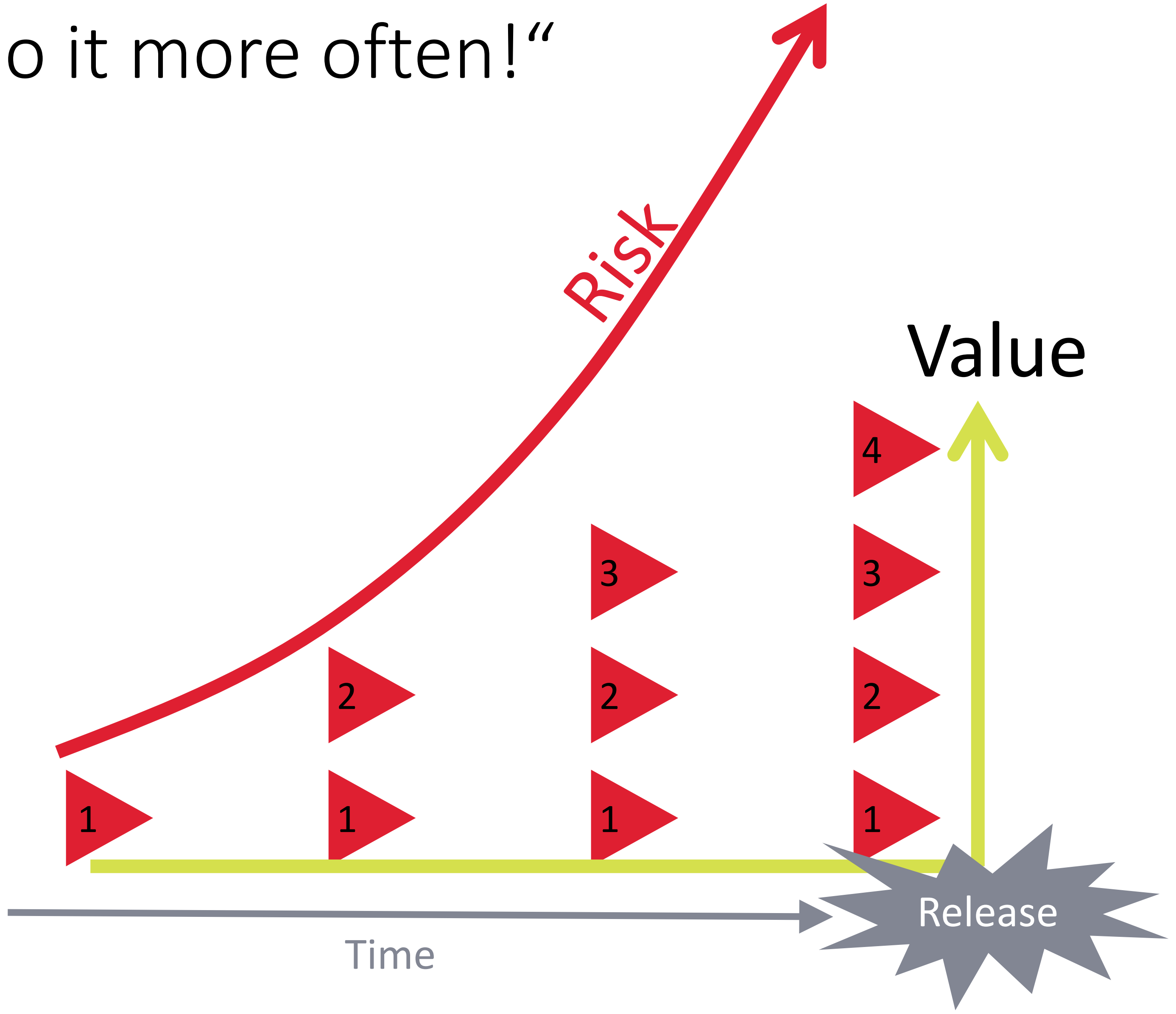
- Keep as-is model up to date
  - Automatic data collection (CMDB, Inventory, …)
- Ensure the EA vision is applied



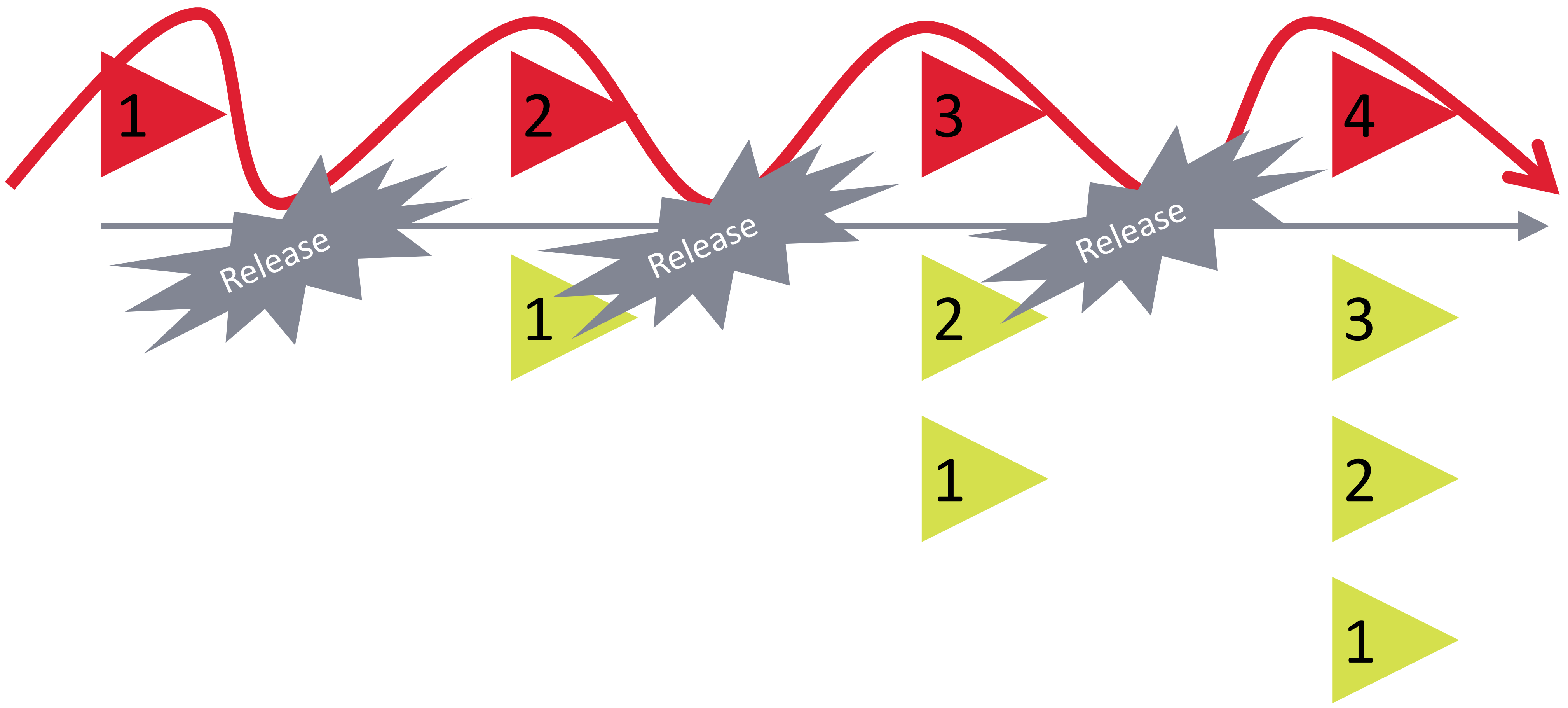
- Consistency



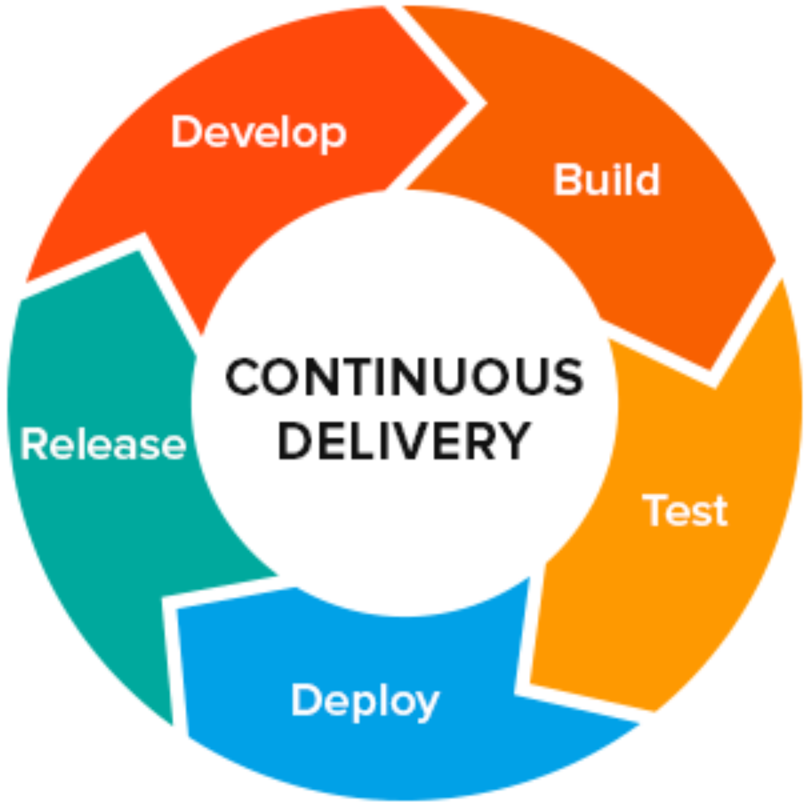
„If it hurts, do it more often!“



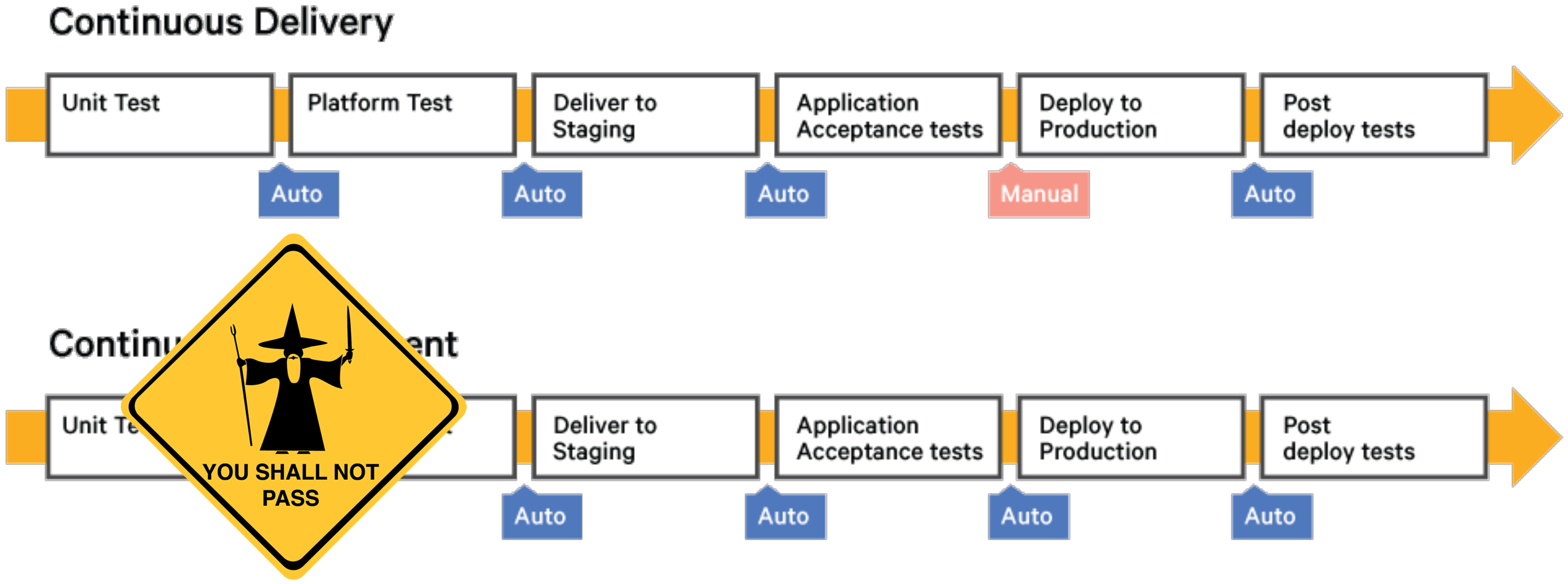
Luca Minudel, 2014



Luca Minudel, 2014



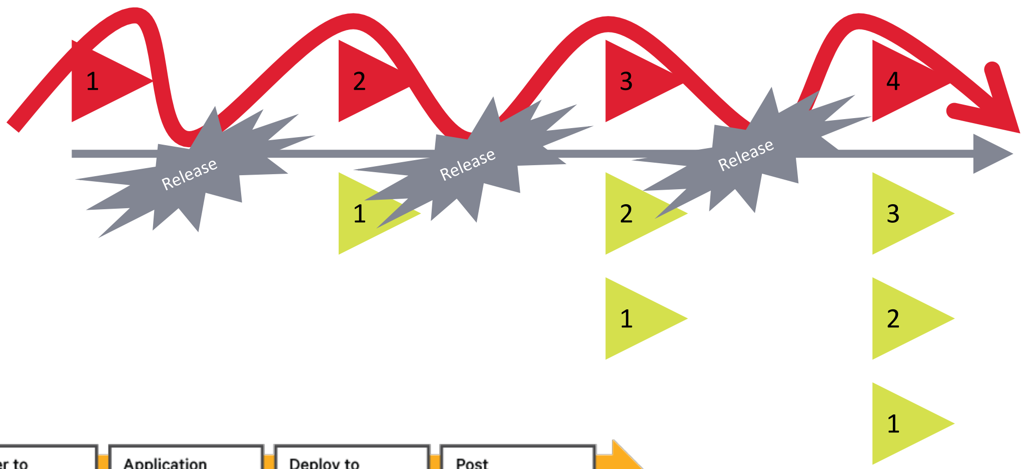
Electric Cloud, 2016



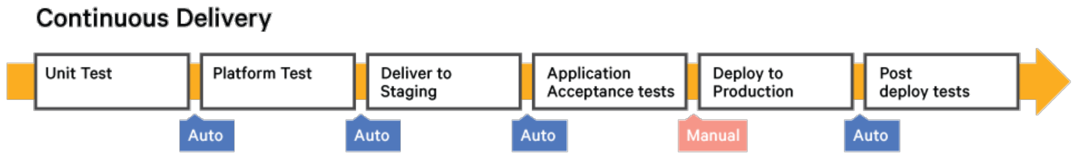
Carl Caum, Puppetlabs, 2013

Continuous Delivery

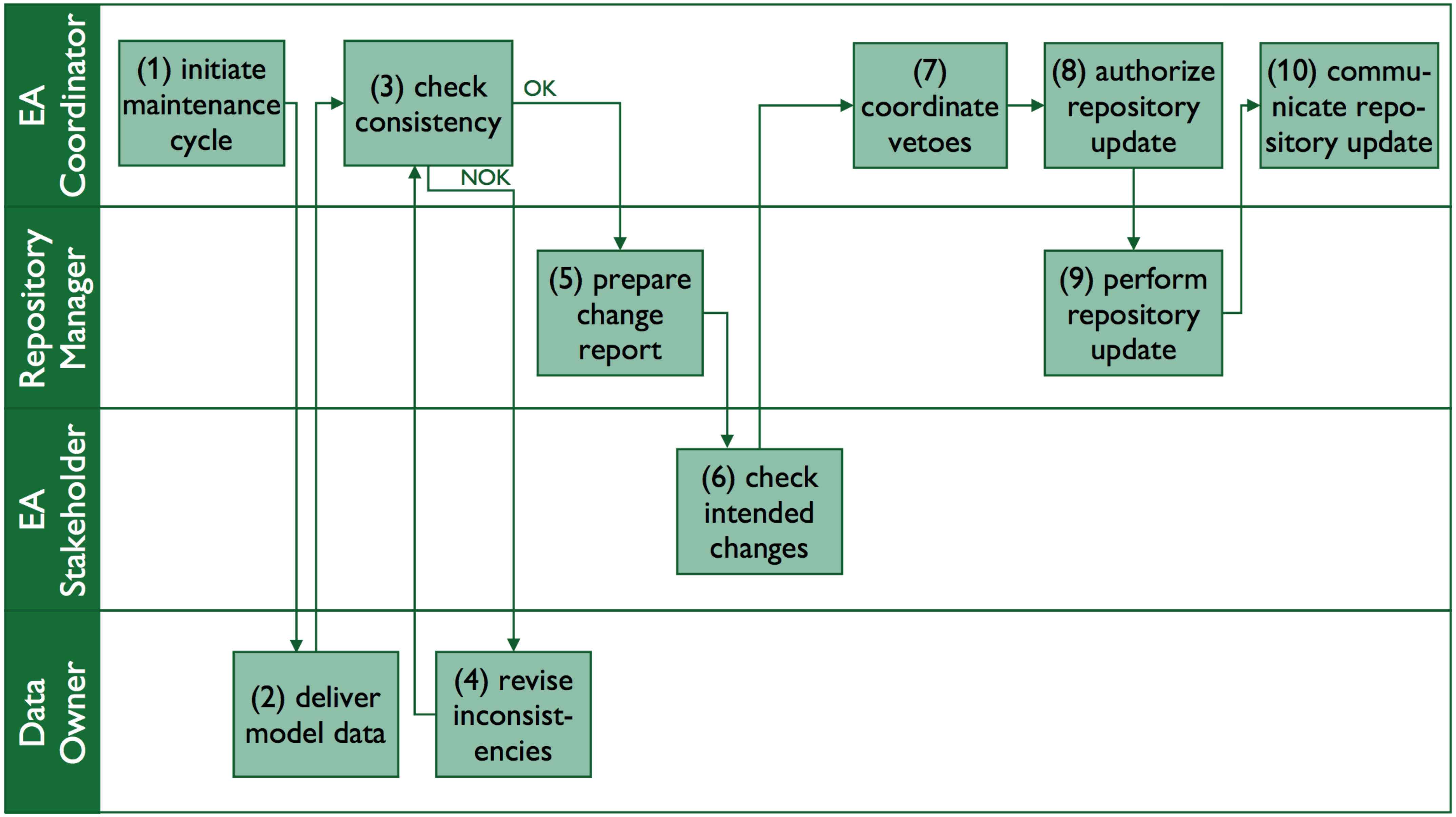
- Risk reduction



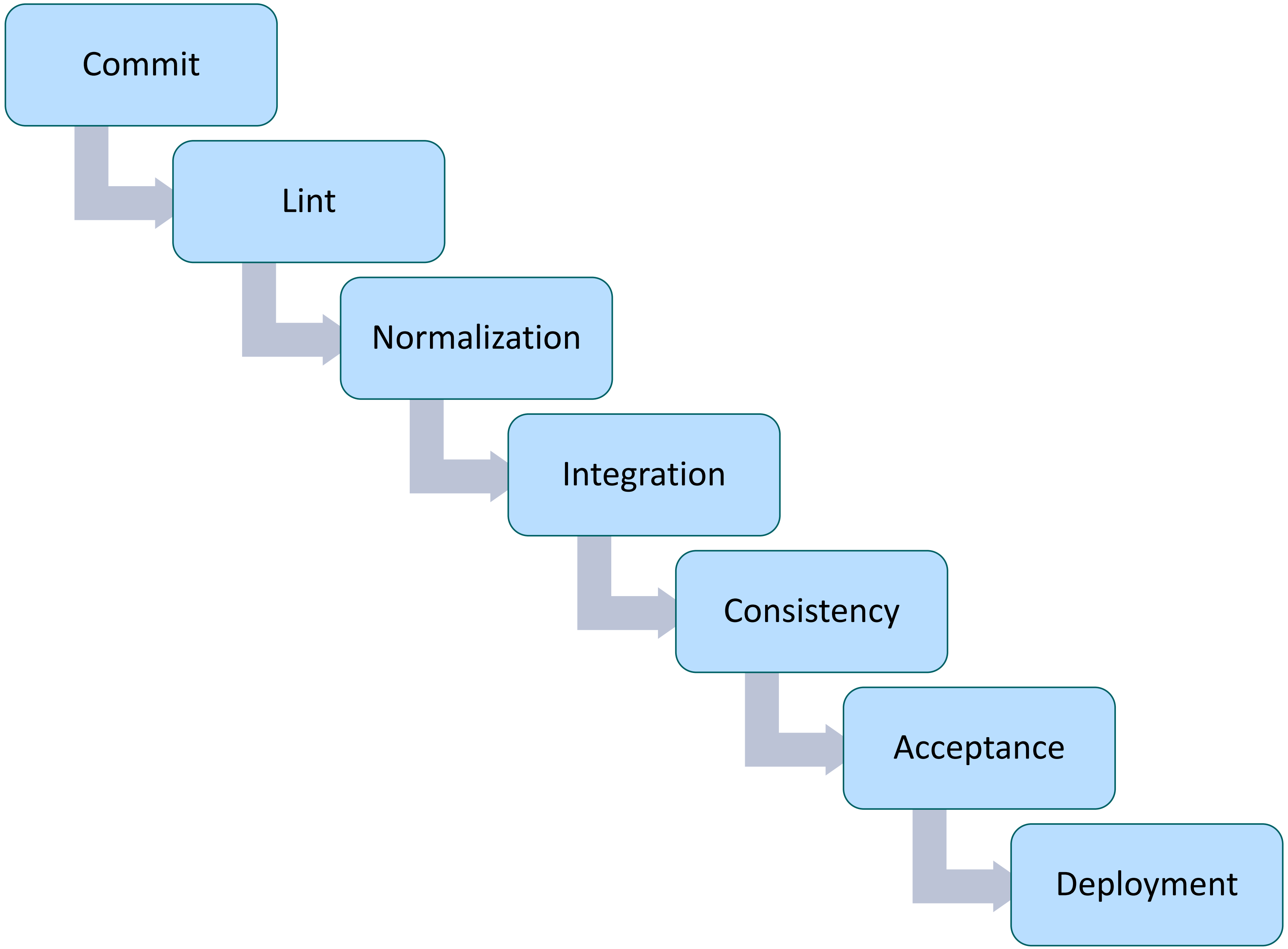
- Pipeline



- Quality gates


EA Coordinator
Repository Manager
EA Stakeholder
Data Owner
(1) initiate maintenance cycle
(2) deliver model data
(3) check consistency
OK
NOK
(4) revise inconsist-encies
(5) prepare change report
(6) check intended changes
(7) coordinate vetoes
(8) authorize repository update
(9) perform repository update
(10) commu-nicate repo-sitory update

Stephan Aier, 2007


Commit
Lint
Normalization
Integration
Consistency
Acceptance
Deployment

Extension: Directed Acceptance

- EA has as-is and to-be state
- Check if as-is is moving towards to-be.

- Insert before manual acceptance

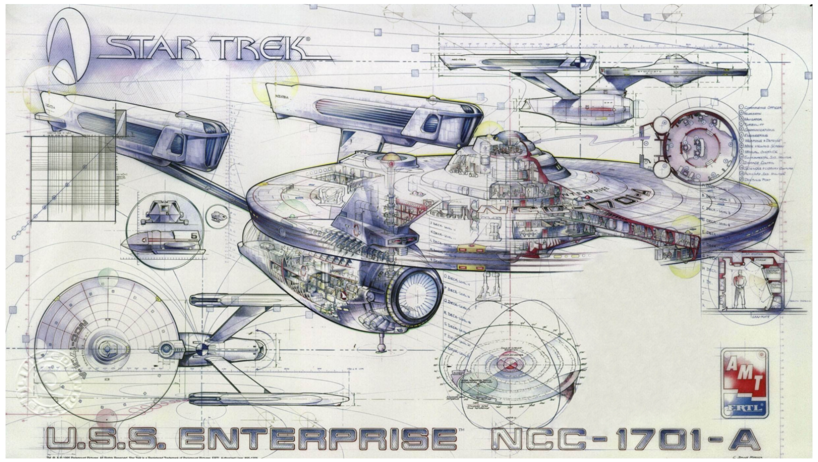
- Keep as-is model up to date
  - Central repository
- Ensure the EA vision is applied
  - Directed Acceptance
- Consistency
  - Consistency check

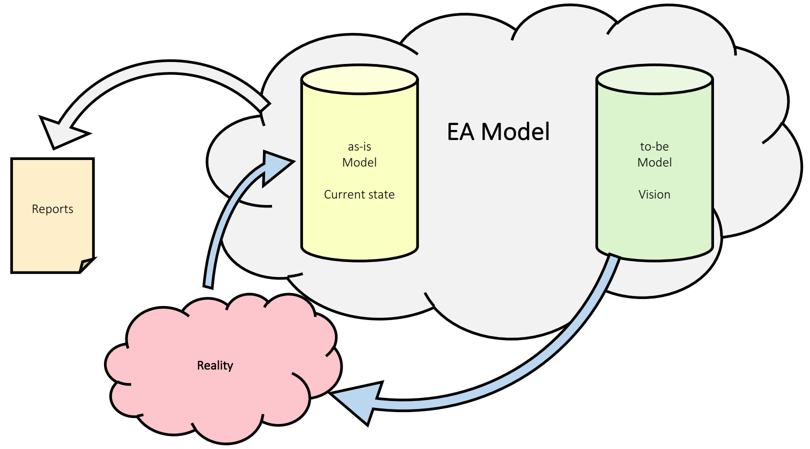
Future work

- Real world implementation
- Understand social and technical impact
- How to define automated acceptance tests?

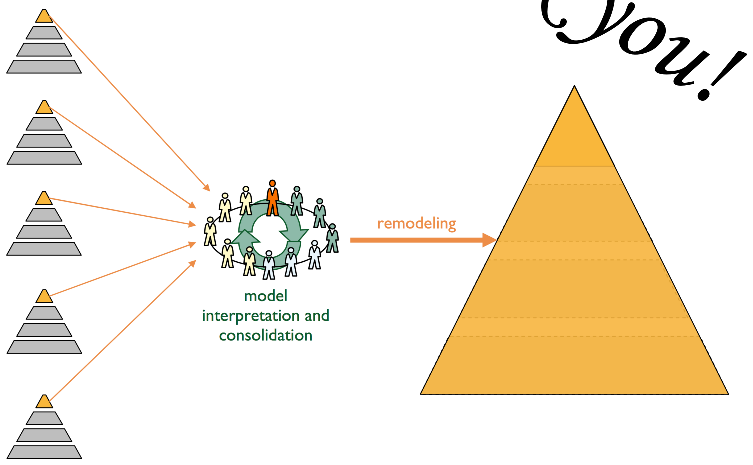
Conclusion

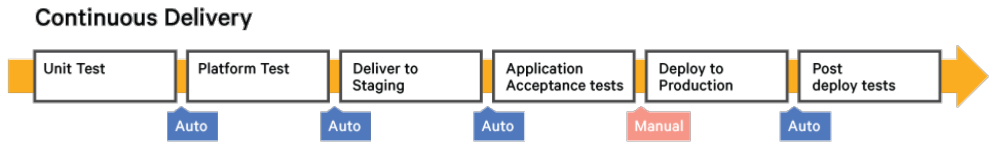
- Process is needed for continuous delivery implementation
- Automation uncovers new aspects
  - Directed acceptance testing
- Manual effort could be reduced


STAR TREK
U.S.S. ENTERPRISE NCC-1701-A

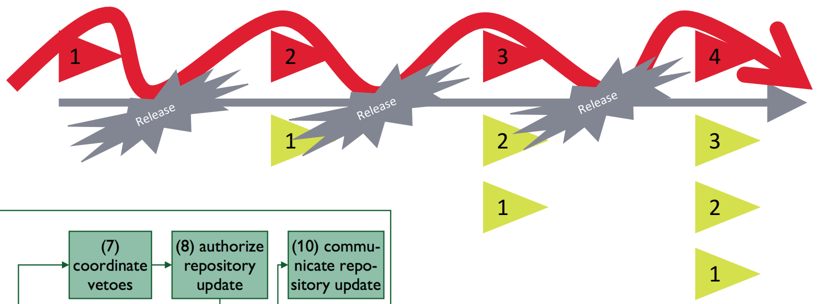

EA Model
as-is
Model
Current state
to-be
Model
Vision
Reports
Reality

Thank you!

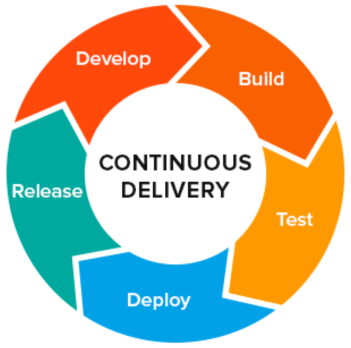

model
interpretation and
consolidation
remodeling

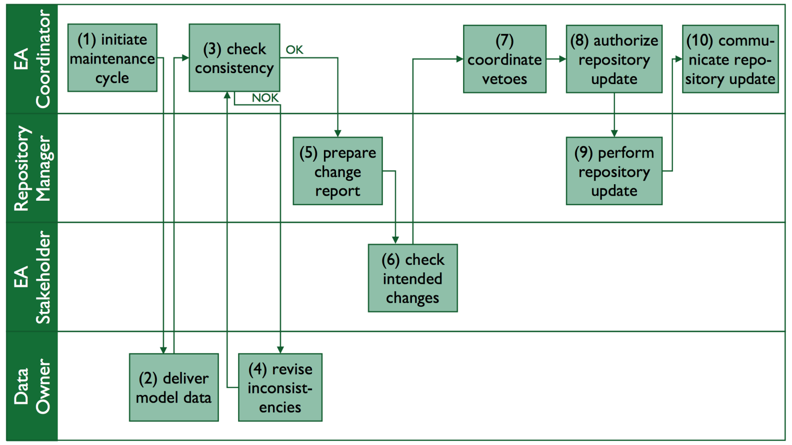

Continuous Delivery
Unit Test
Platform Test
Deliver to
Staging
Application
Acceptance tests
Deploy to
Production
Post
deploy tests
Auto
Auto
Auto
Manual
Auto

YOU SHALL NOT
PASS


1
2
3
4
Release
Release
Release
1
2
3
1
2
1

WRONG
WAY
GO BACK


Develop
Build
Test
Deploy
Release
CONTINUOUS
DELIVERY


EA Coordinator
Repository Manager
EA Stakeholder
Data Owner
(1) initiate maintenance cycle
(2) deliver model data
(3) check consistency
OK
NOK
(4) revise inconsist-encies
(5) prepare change report
(6) check intended changes
(7) coordinate vetoes
(8) authorize repository update
(9) perform repository update
(10) commu-nicate repo-sitory update

Maximilian Peiffer
Maximilian.Peiffer@rwth-aachen.de

From EA models to UML

A guideline

Everything is about plans

Business Plans

Plans for Shareholders

Plans for Business

Enterprise Architecture

UML

The problem:
Only EA models

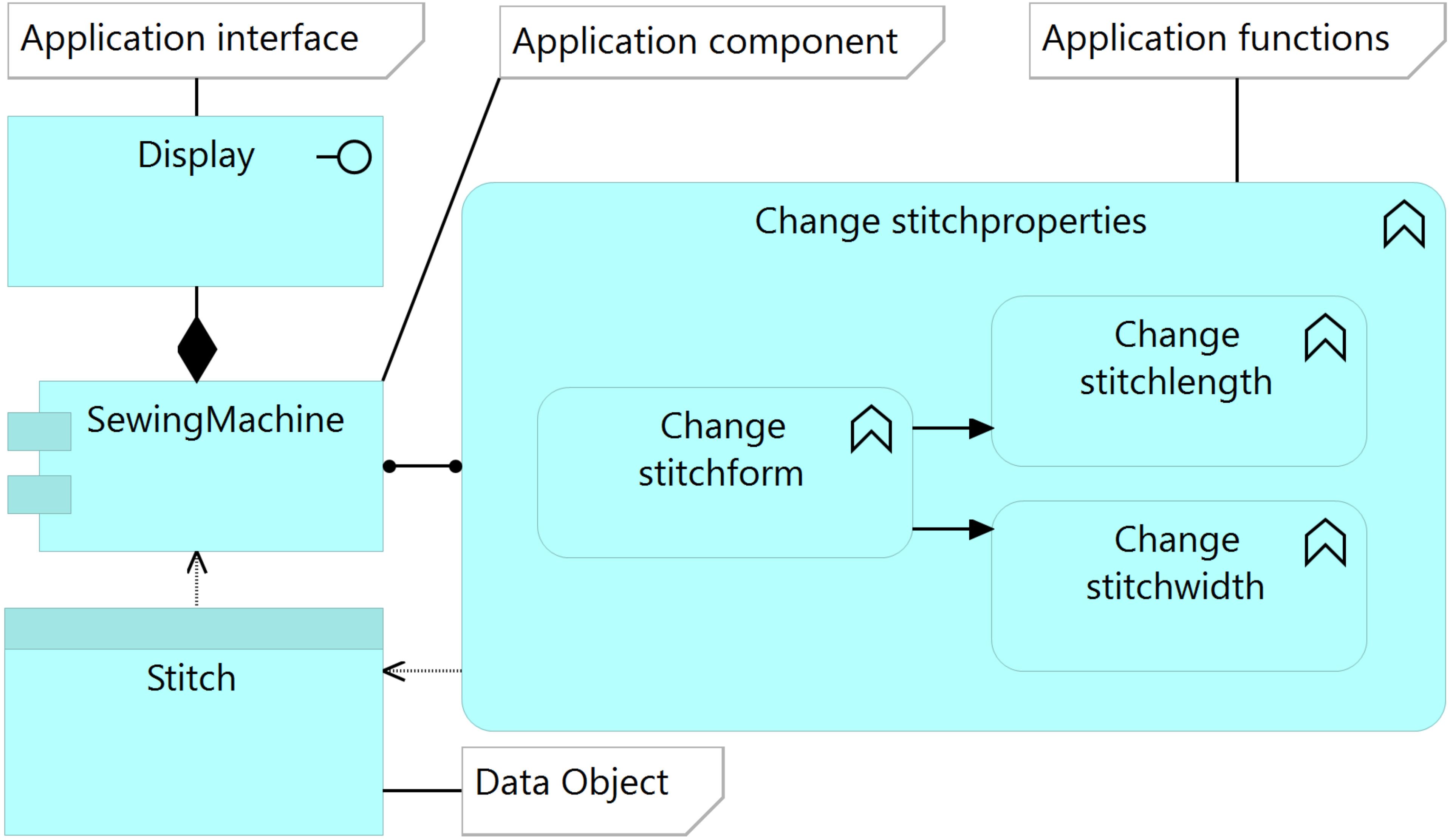
The solution:
Create UML models

Transformation

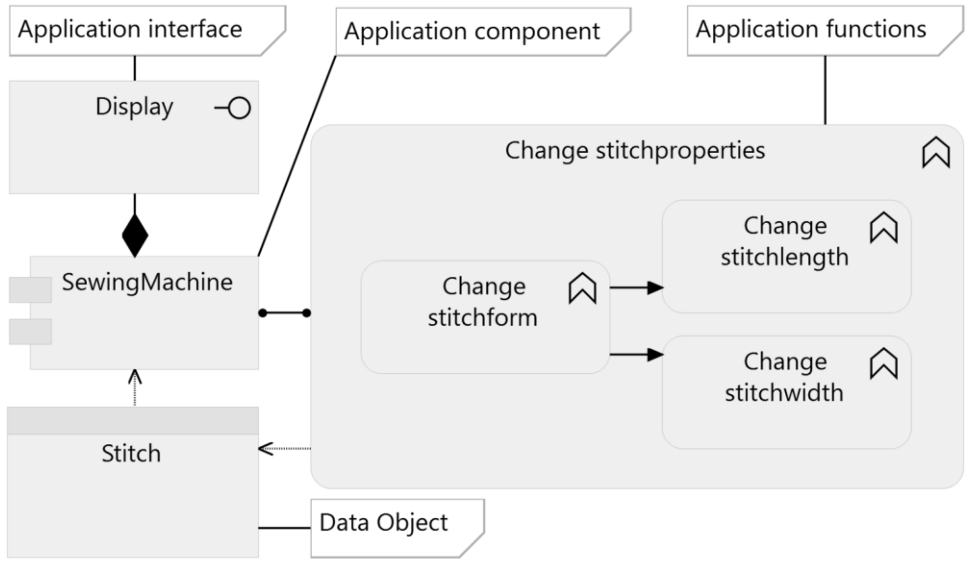
The new problem:
How to transform?

The solution:
A guideline

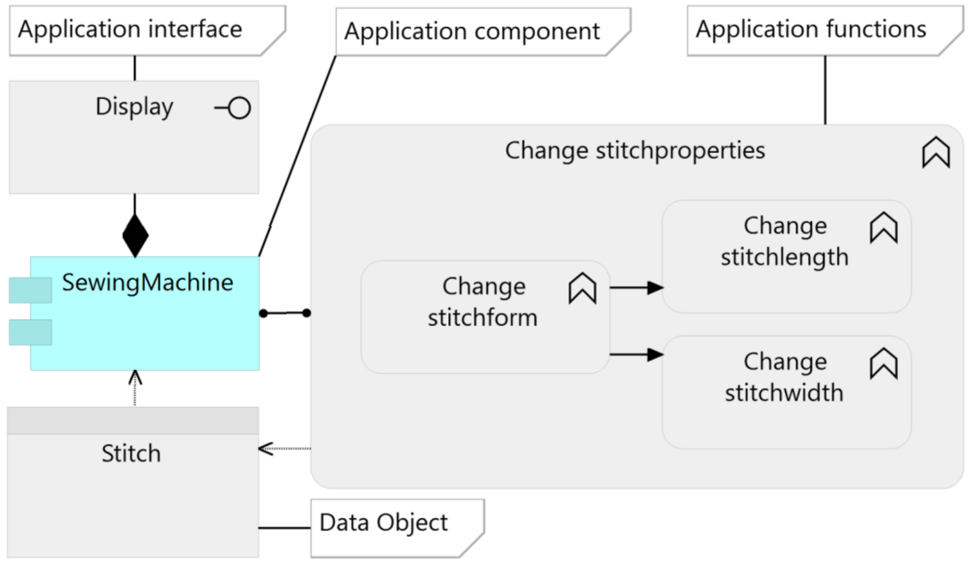
A Guideline

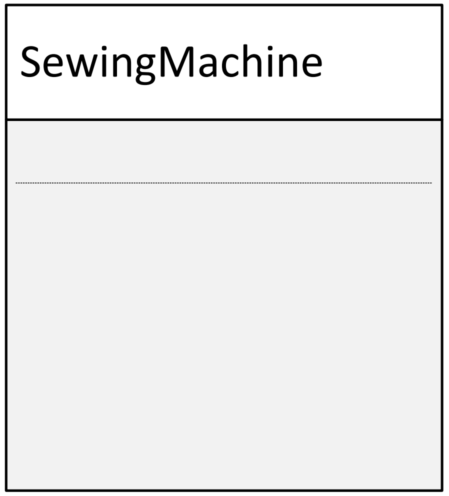


Example: ArchiMate

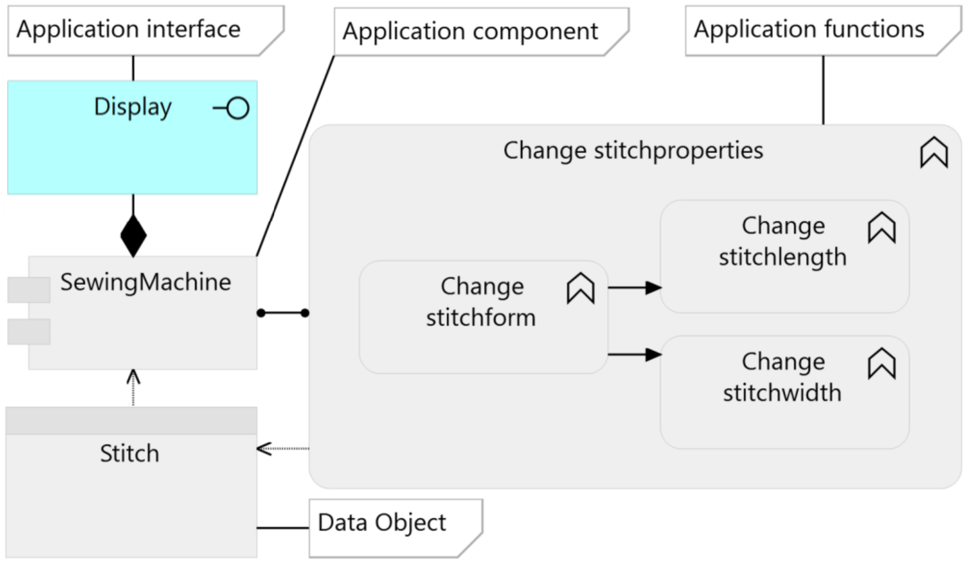


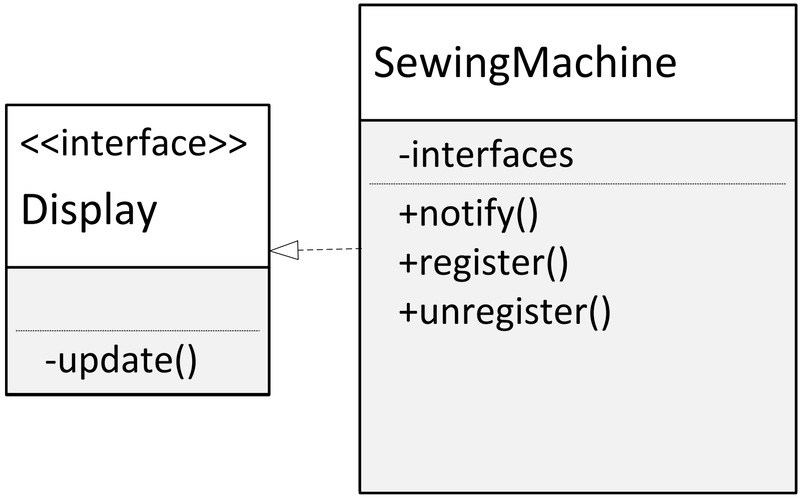
Example: Class Diagram



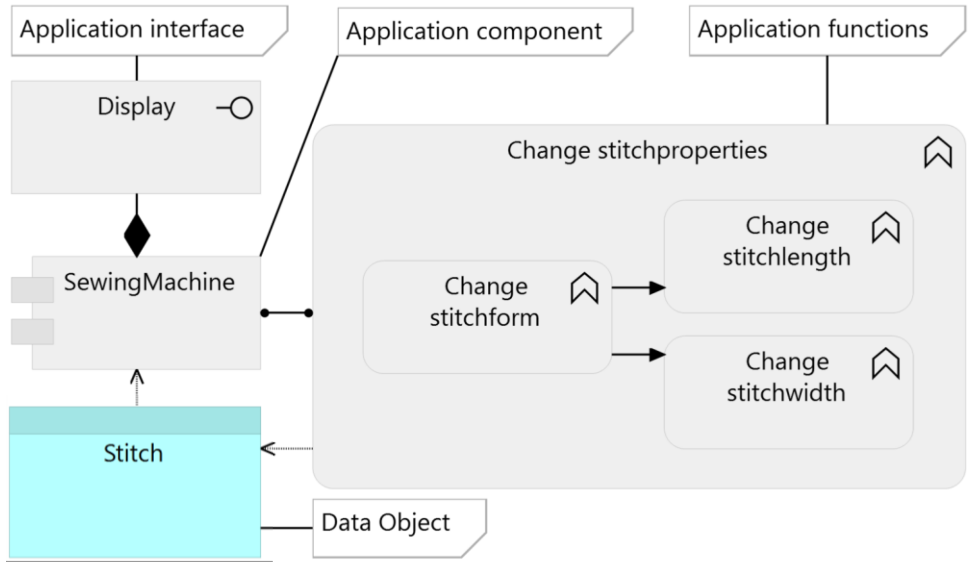


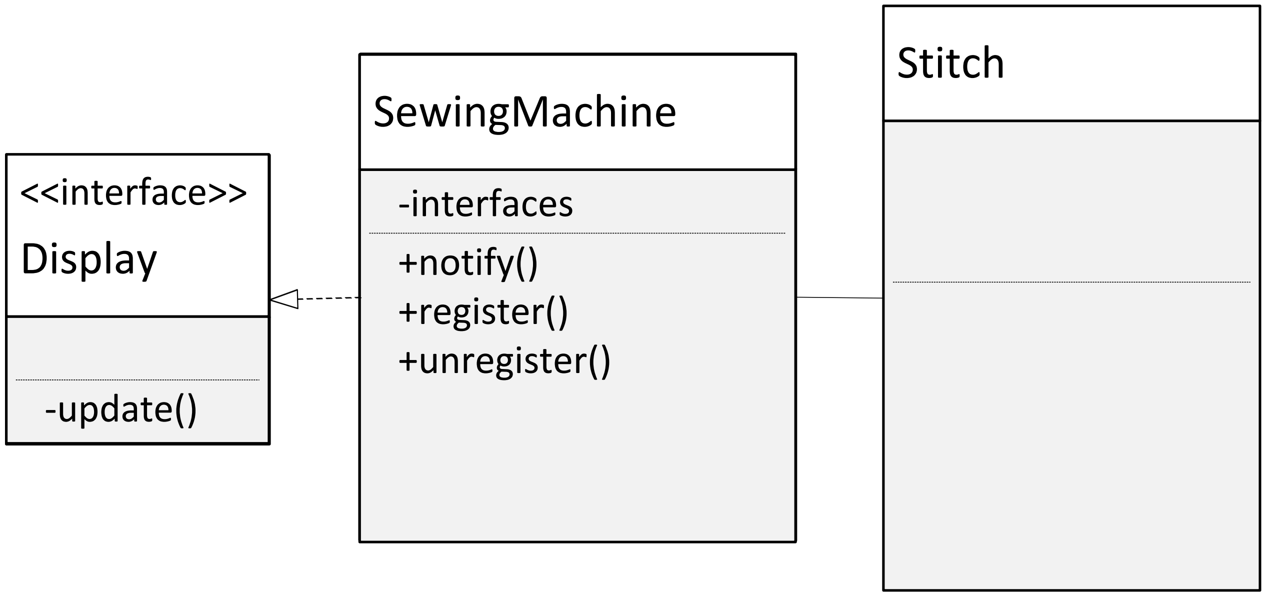
Example: Class Diagram



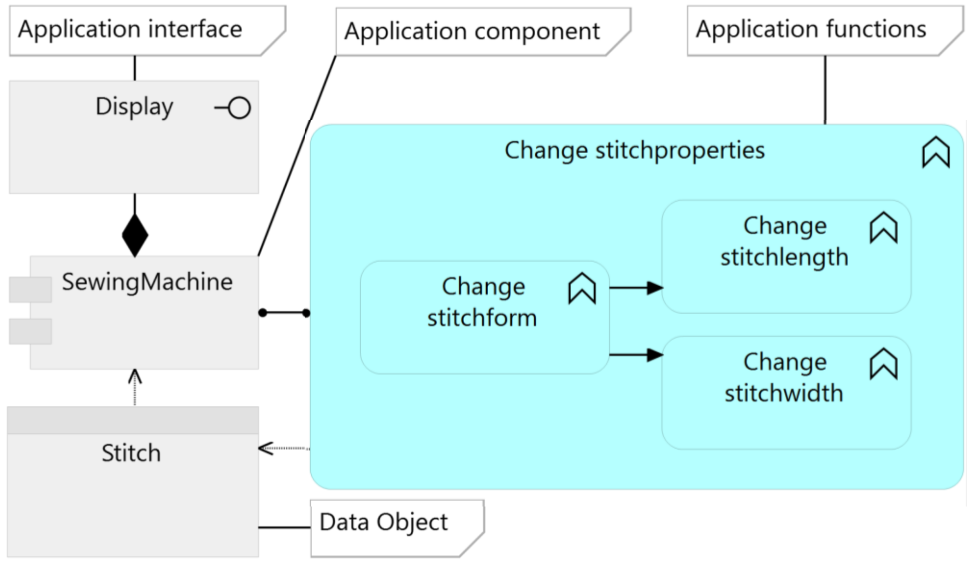


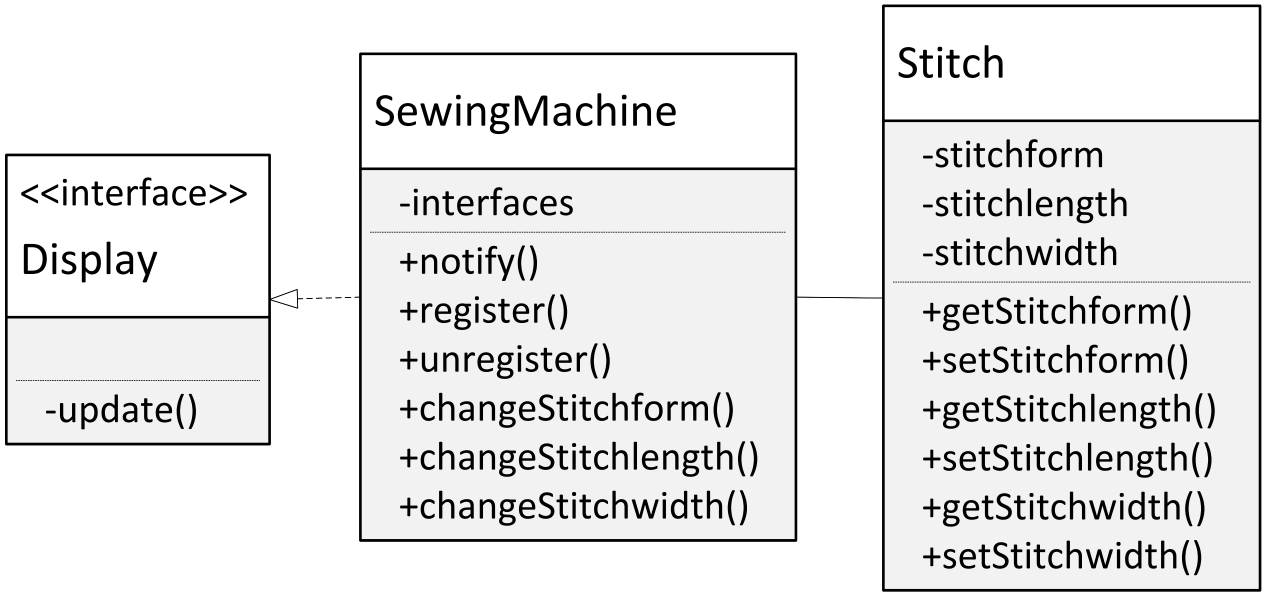
Example: Class Diagram



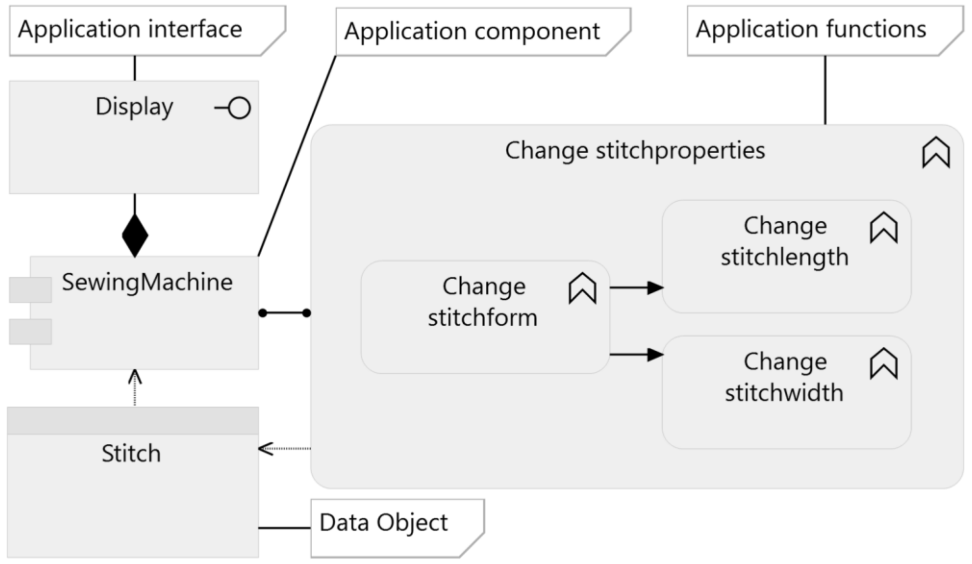


Example: Class Diagram

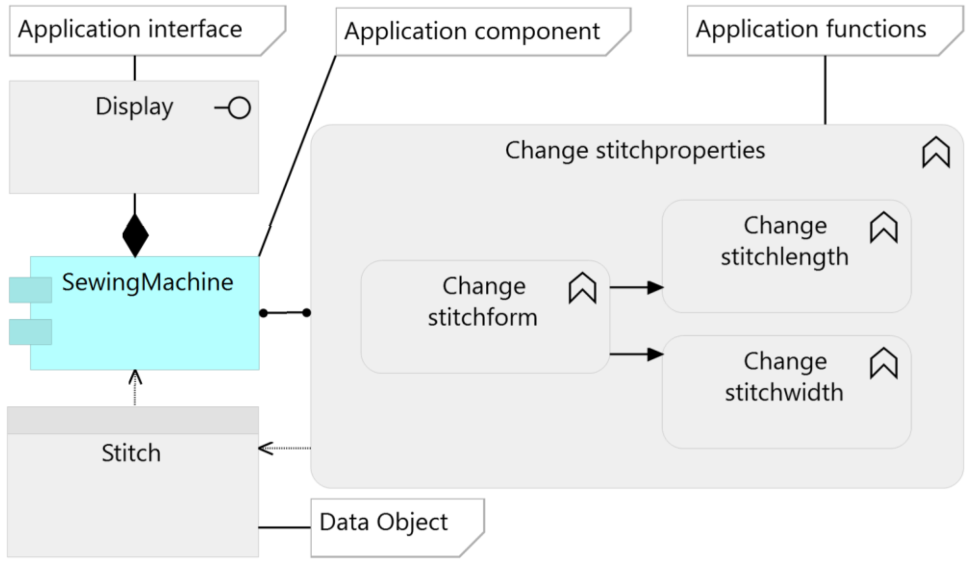


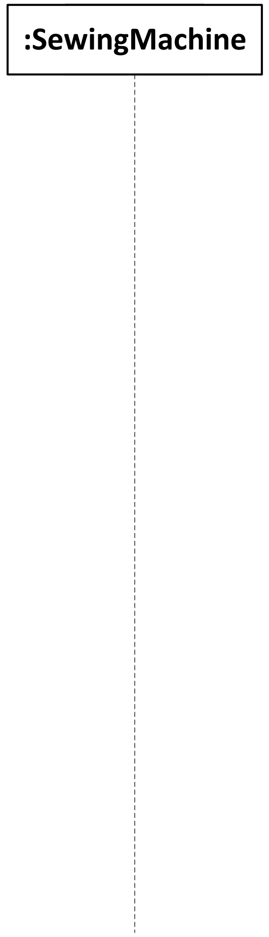


Example: Class Diagram

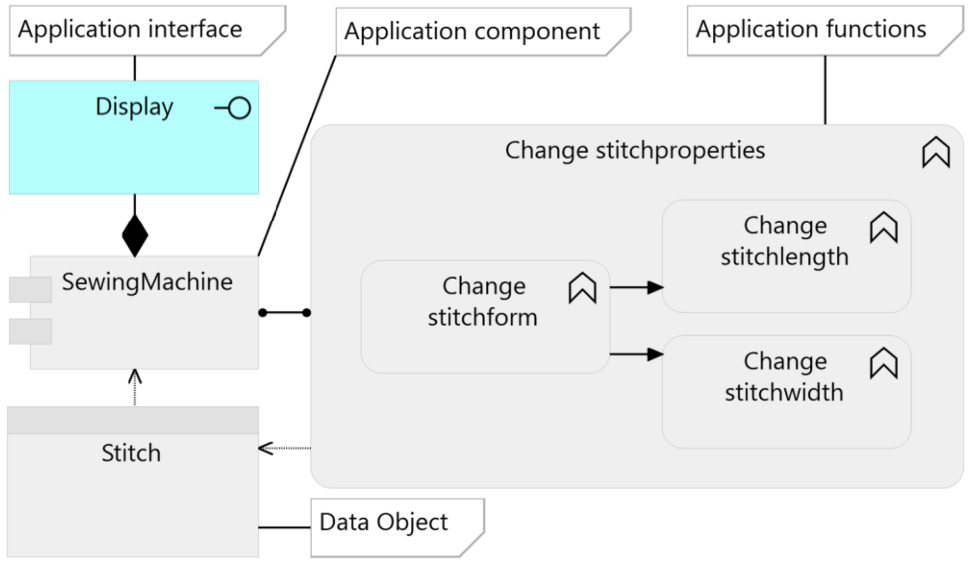


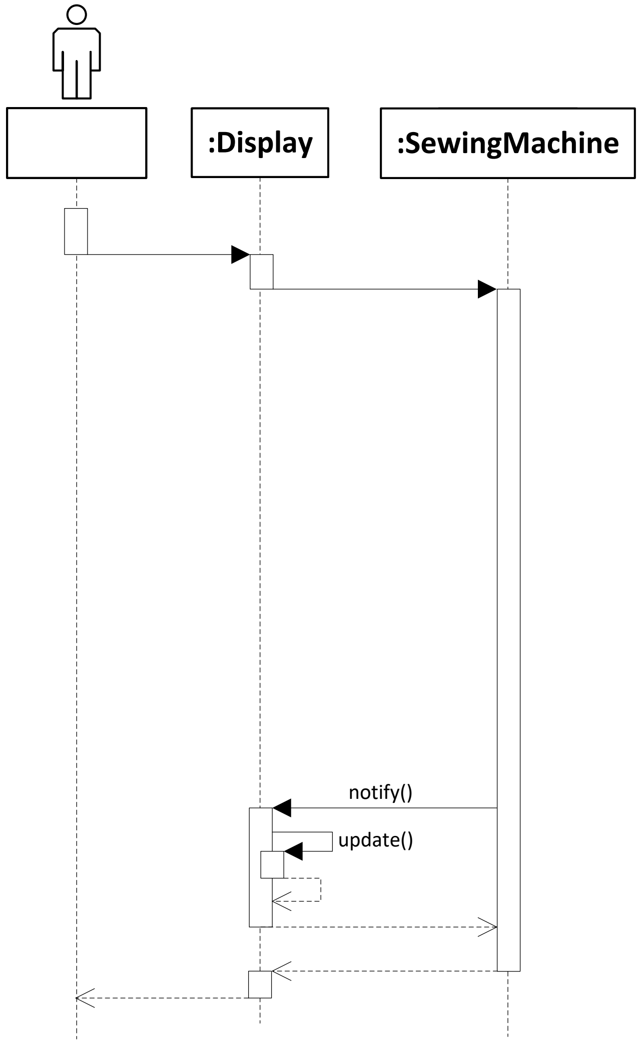
Example: Sequene Diagram



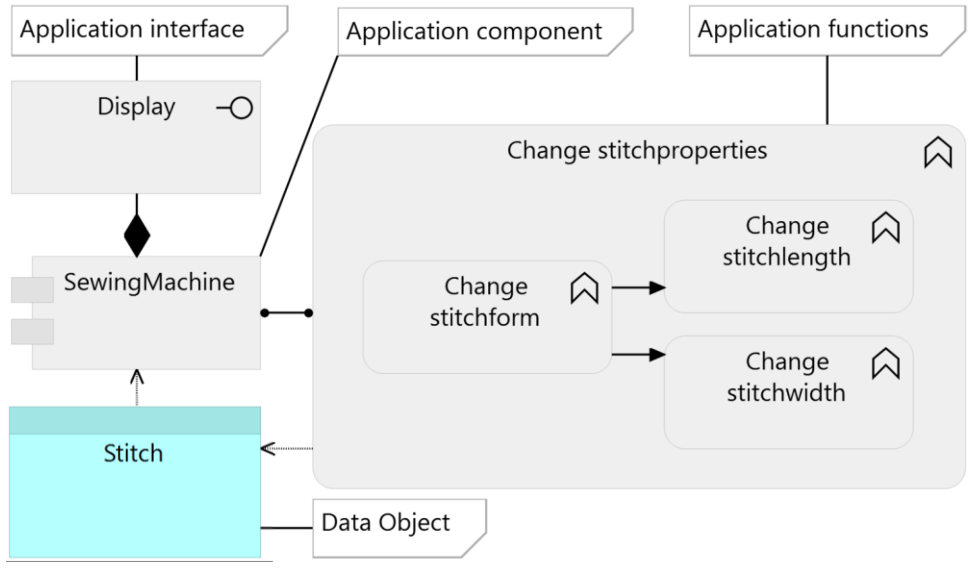


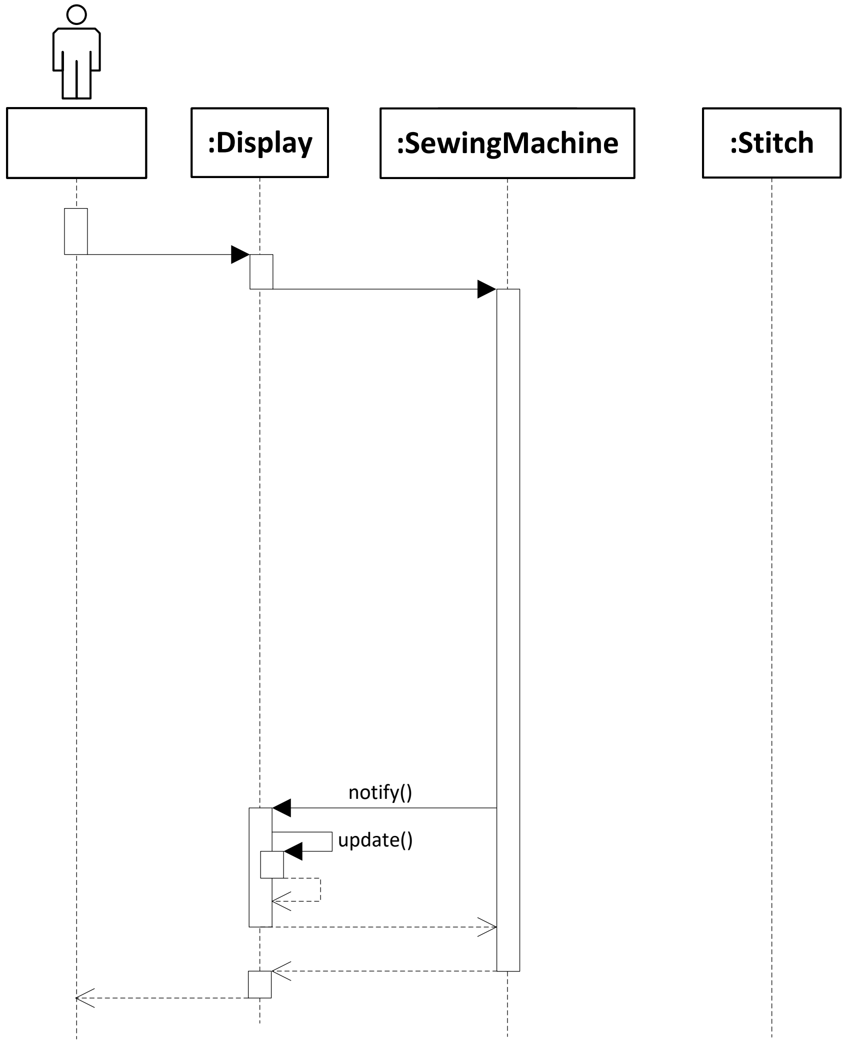
Example: Sequene Diagram



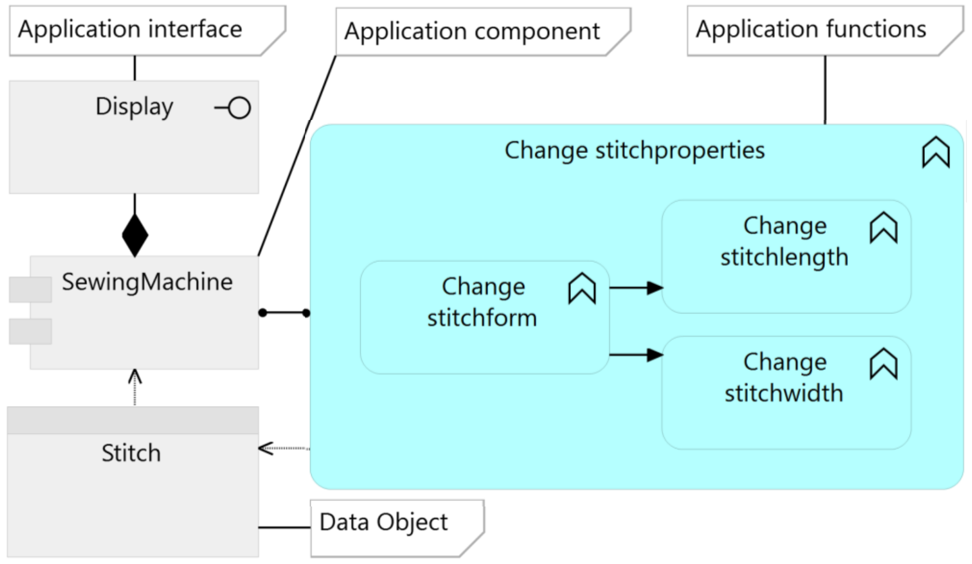


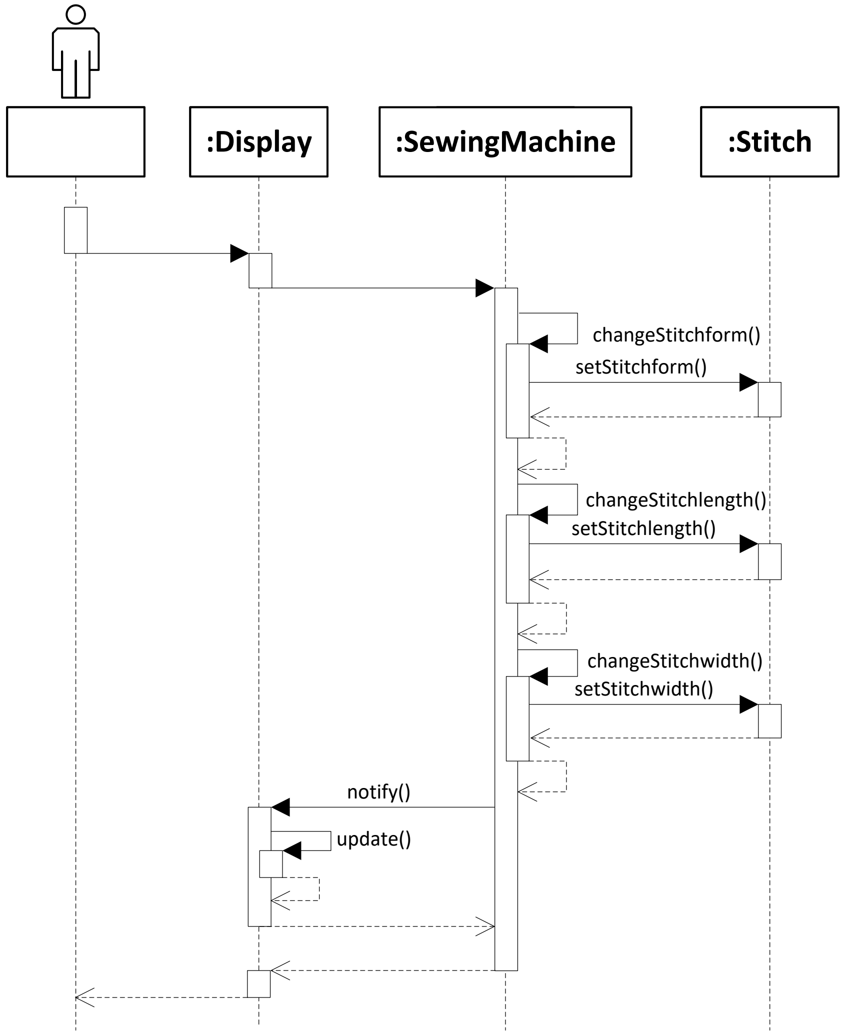
Example: Sequene Diagram



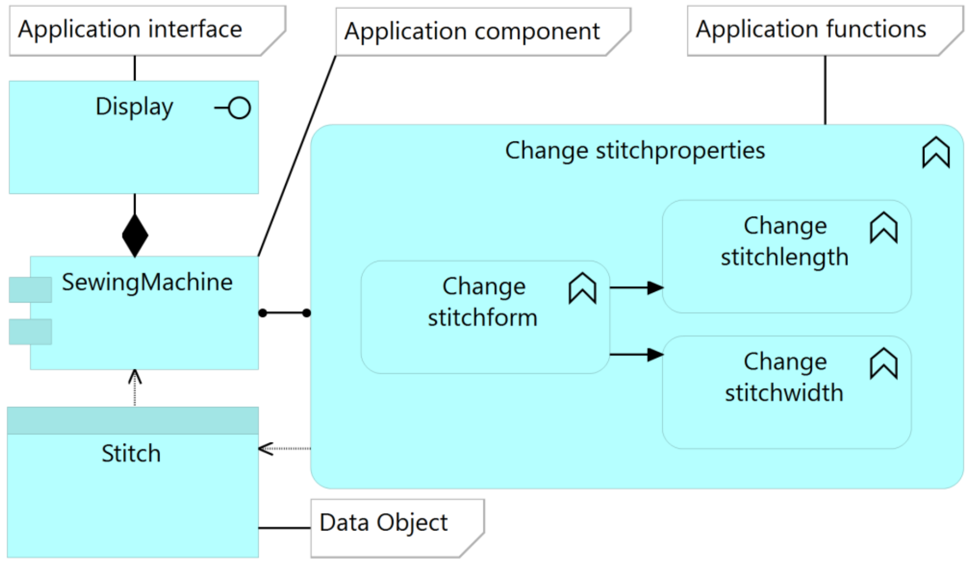


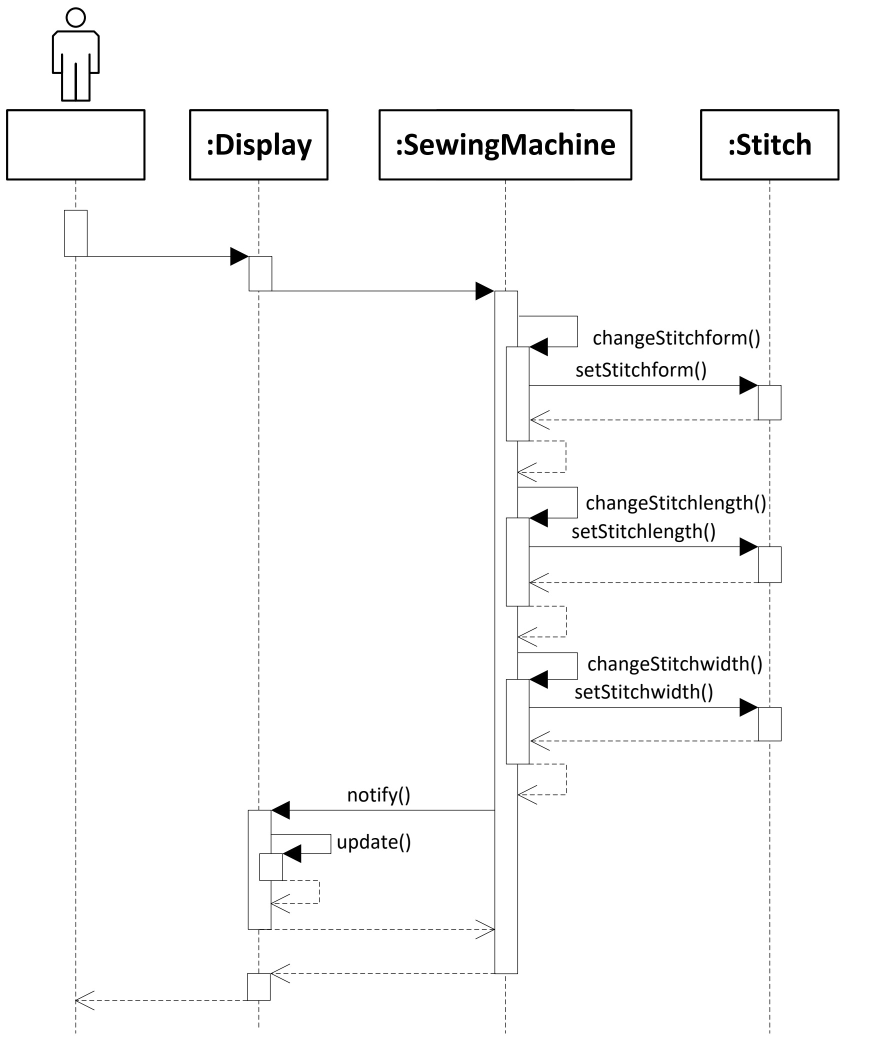
Example: Sequene Diagram

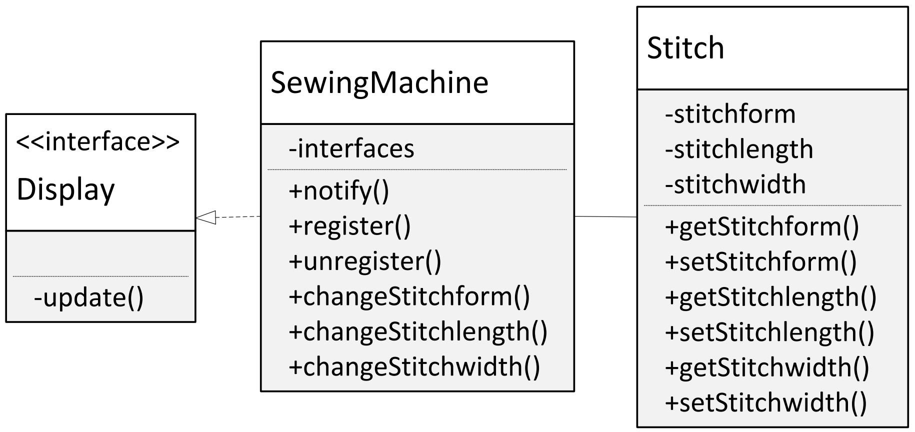




Example: Sequene Diagram


Application interface
Application component
Application functions
Display
Change stitchproperties
Change stitchlength
Change stitchform
SewingMachine
Change stitchwidth
Stitch
Data Object

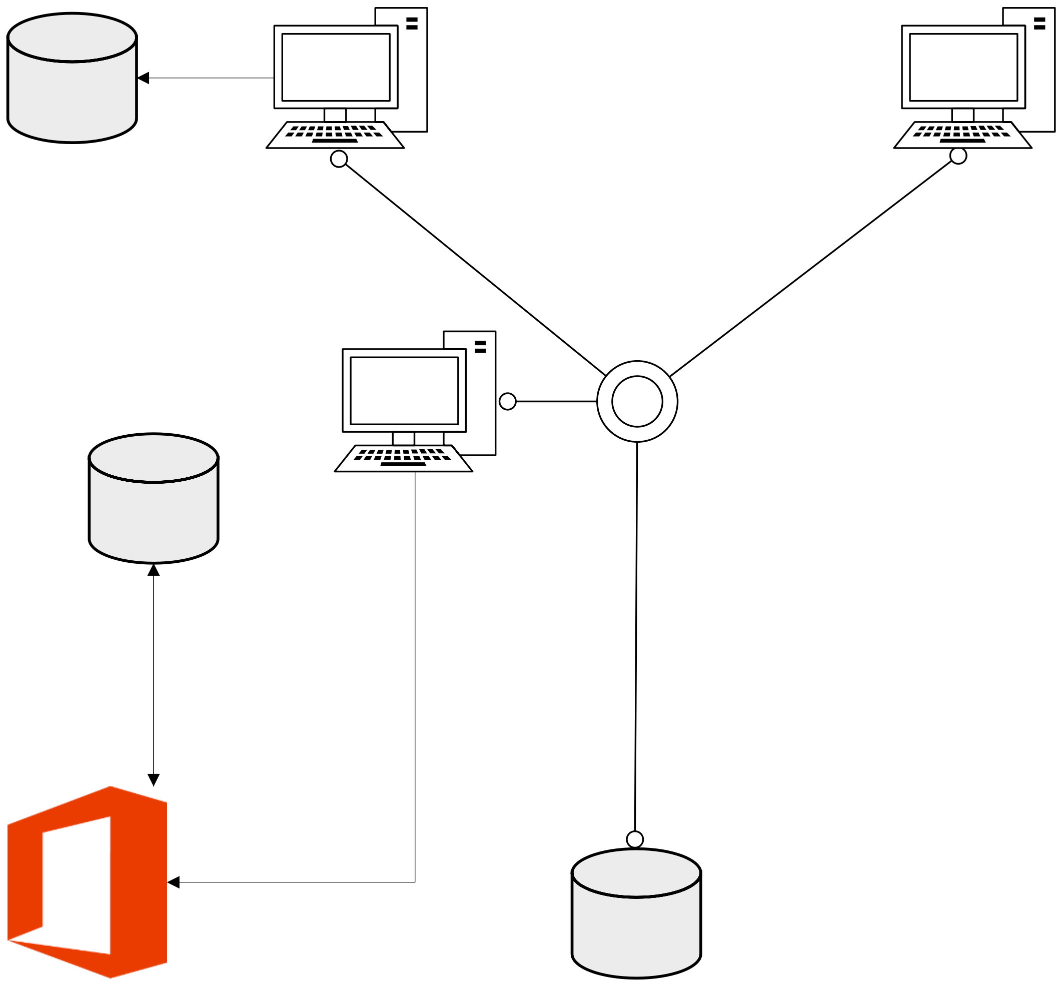

:Display
:SewingMachine
:Stitch
changeStitchform()
setStitchform()
changeStitchlength()
setStitchlength()
changeStitchwidth()
setStitchwidth()
notify()
update()


<<interface>>
Display
-update()
SewingMachine
-interfaces
+notify()
+register()
+unregister()
+changeStitchform()
+changeStitchlength()
+changeStitchwidth()
Stitch
-stitchform
-stitchlength
-stitchwidth
+getStitchform()
+setStitchform()
+getStitchlength()
+setStitchlength()
+getStitchwidth()
+setStitchwidth()

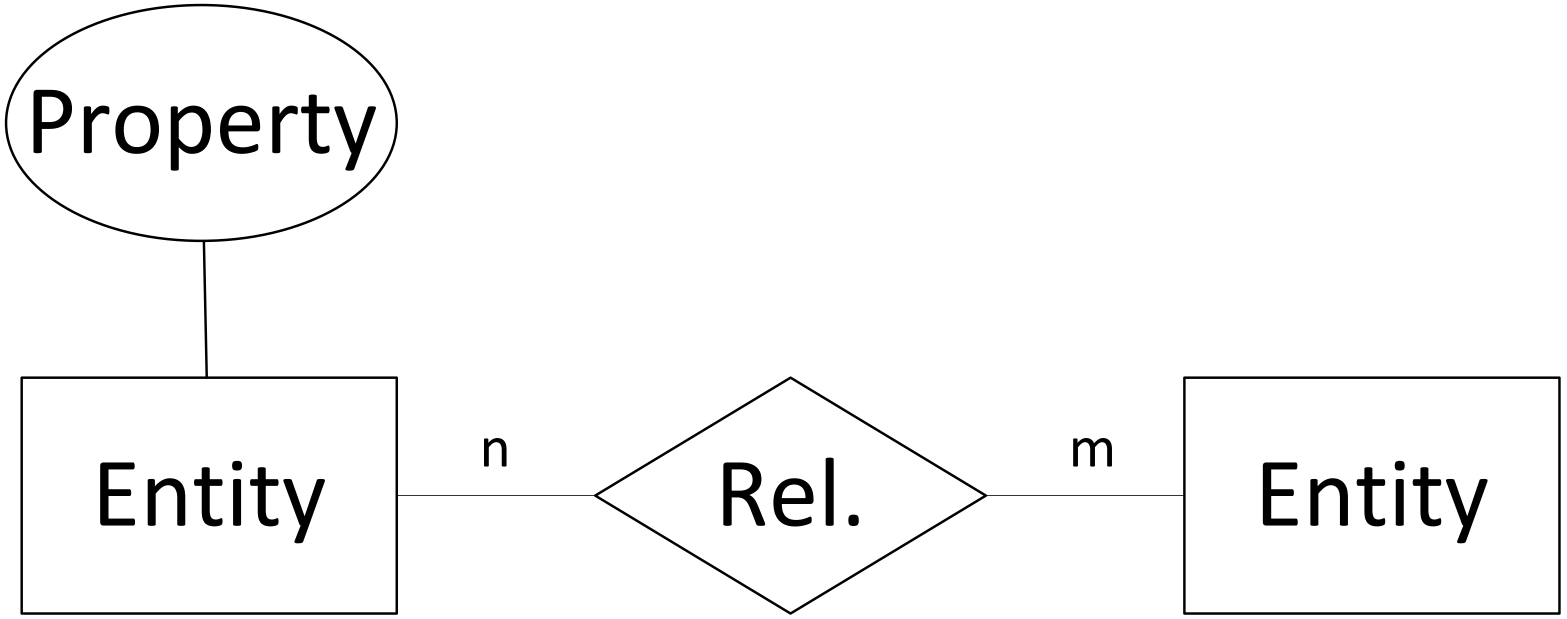
The new problem:
Lack of information

The solution:
Additional Sources

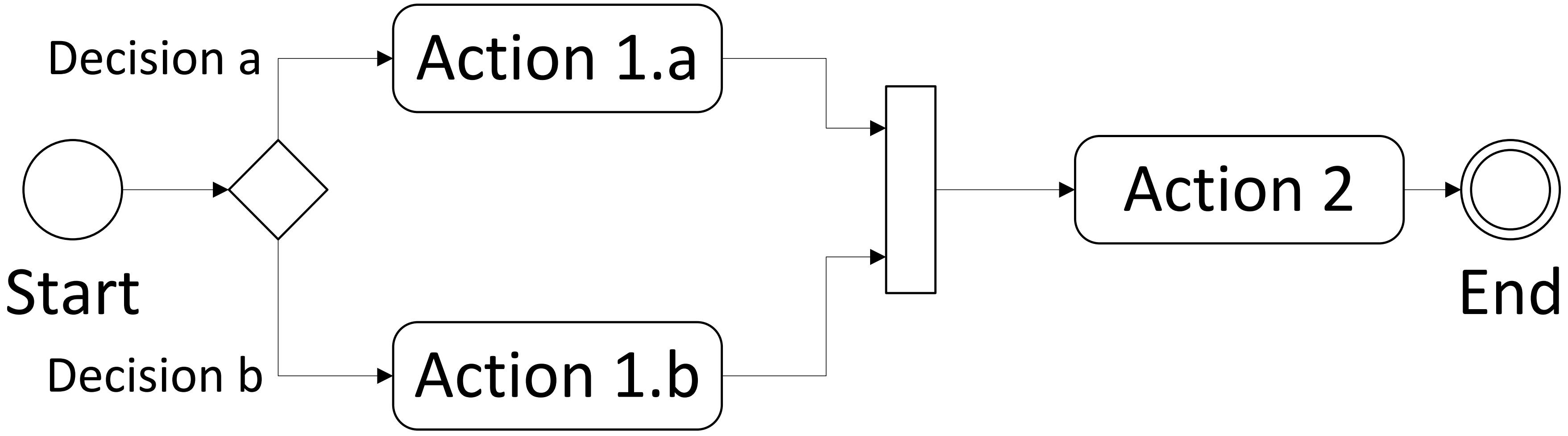
Additional Sources

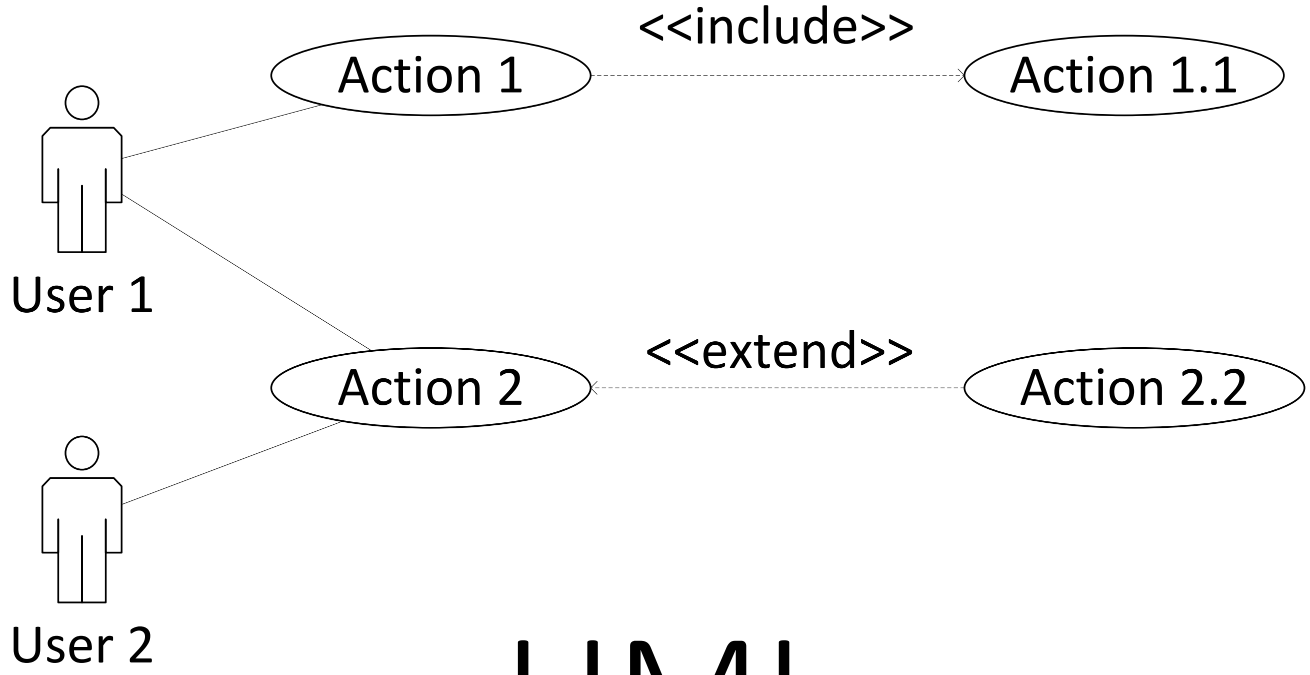


CMDB



ER


Decision a
Action 1.a
Start
End
Action 2
Decision b
Action 1.b


<<include>>
Action 1
Action 1.1
User 1
<<extend>>
Action 2
Action 2.2
User 2

UML

Everything is about plans

Thank you

Transformation

A Guideline

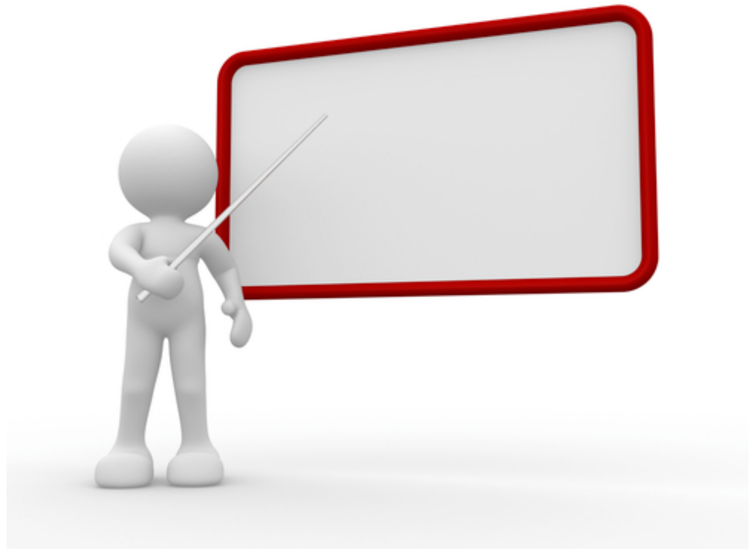
Additional Sources

Joel Hermanns,
joel.hermanns@rwth-aachen.de

Current state of best practices for developing automated tests

Outline

- Introduction
- Influencing Factors
- Existing Best Practices
- Missing Best Practices
- Summary



- Properties of an automated test
  - Validity
  - Determinism and Repeatability
  - Independence

- Best Practice

“a working method or set of working methods that is officially accepted as being the best to use in a particular business or industry, usually described formally and in detail” [3]

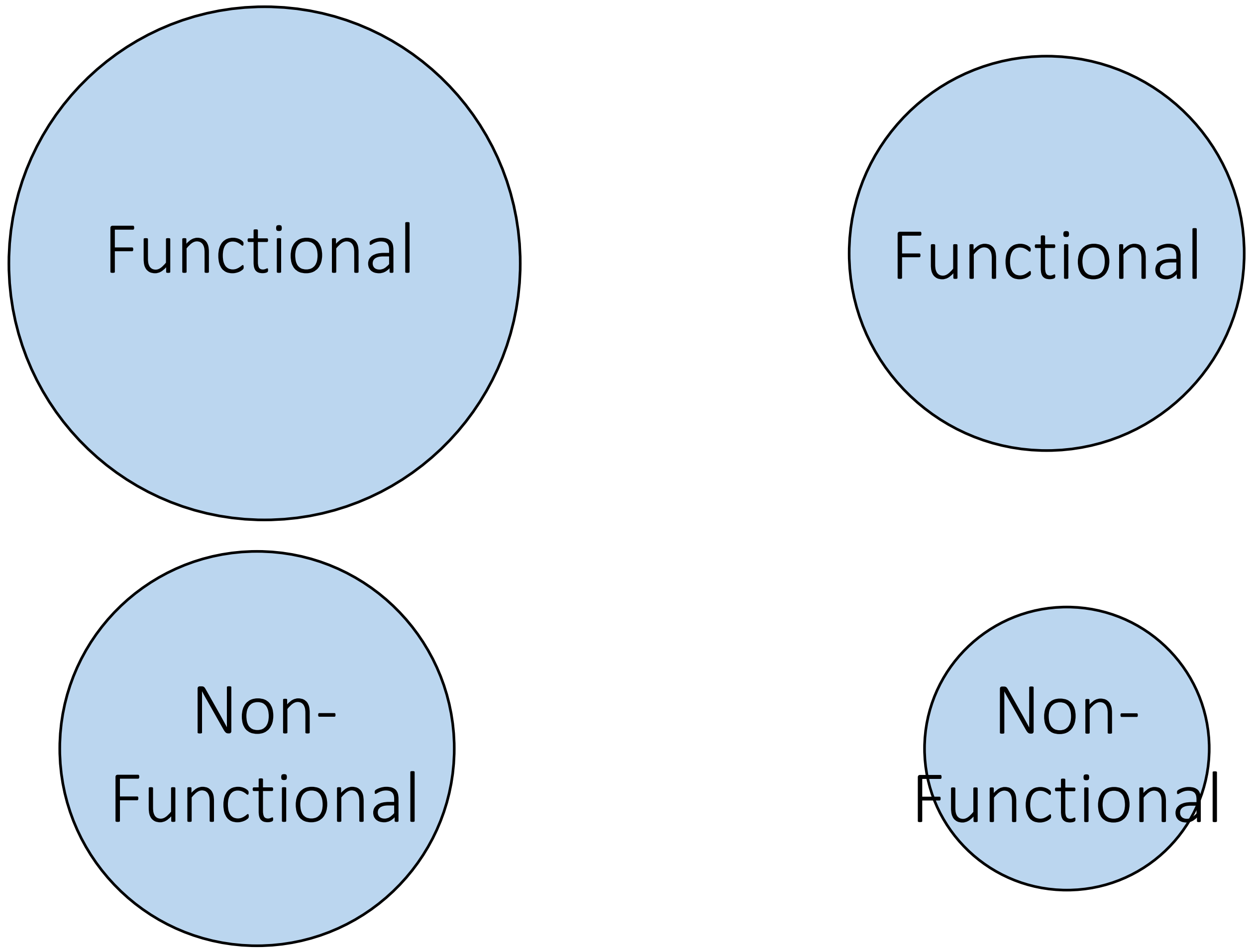
Influencing Factors

- The **level** of testing
  - Unit level
  - Systems level
- The **type** of testing
  - functional
  - non-functional
- **Environment** (e.g. deployment pipeline, development machine)
- **Technology**
  - Programming Language, Libraries, Frameworks, Tools
  - Architecture
  - Access to the system

Existing Best Practices - Overview

Unit Level

Systems Level



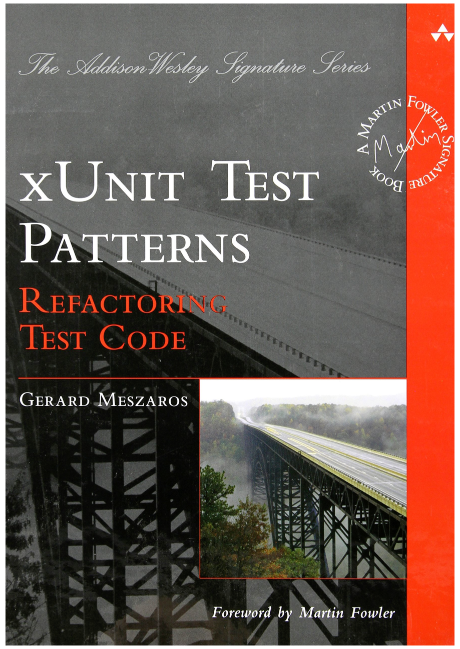
Existing Best Practices - Overview

- Technology-wise: Most best practices can be found in the Web domain
- Programming language: Focus on rather high level, esp. Object Oriented languages
- Access to the system: Very little literature where access is rather complicated

Existing Best Practices – Unit Level

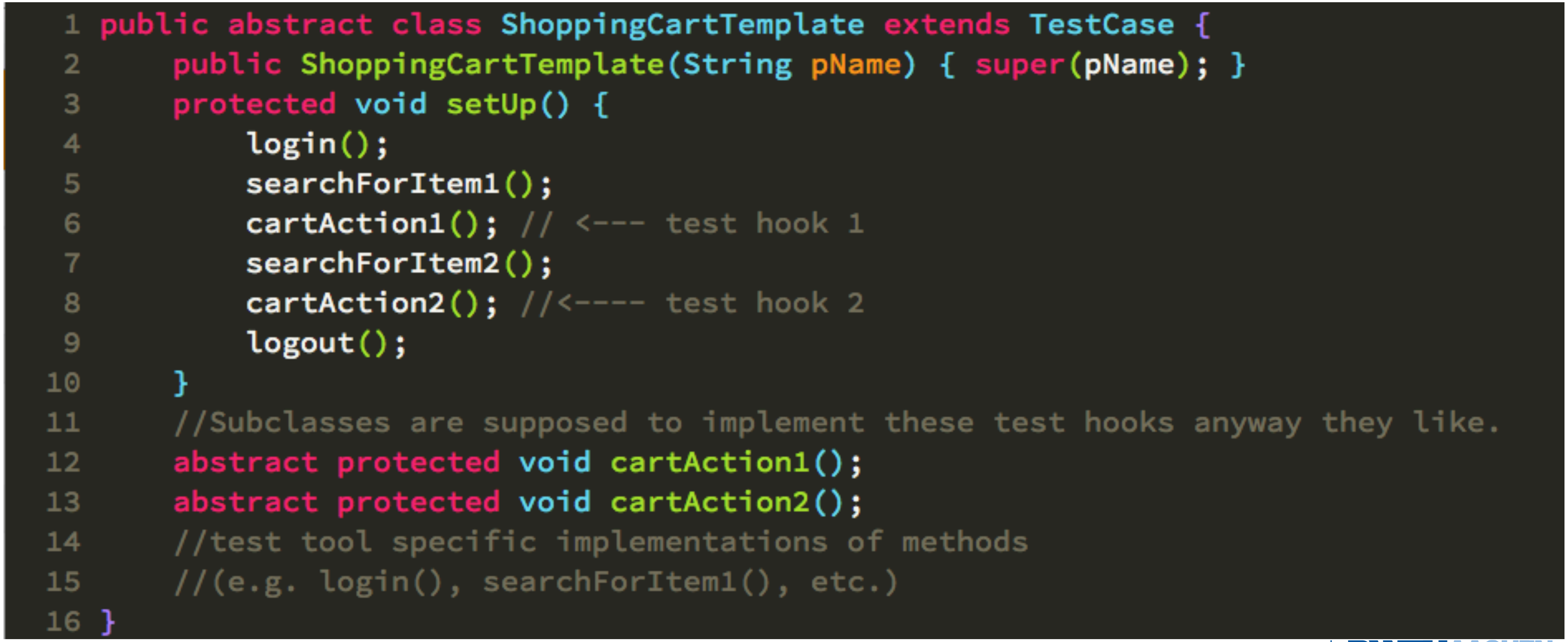
- Various resources for Test Design Patterns in General
  - Very interesting on Unit Level: MockObject Pattern

- Framework/Libraries usually provide individual best practices:
  - E.g. Web frameworks: Ruby on Rails, Spring Framework


The Addison-Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler

Existing Best Practices – Systems Level

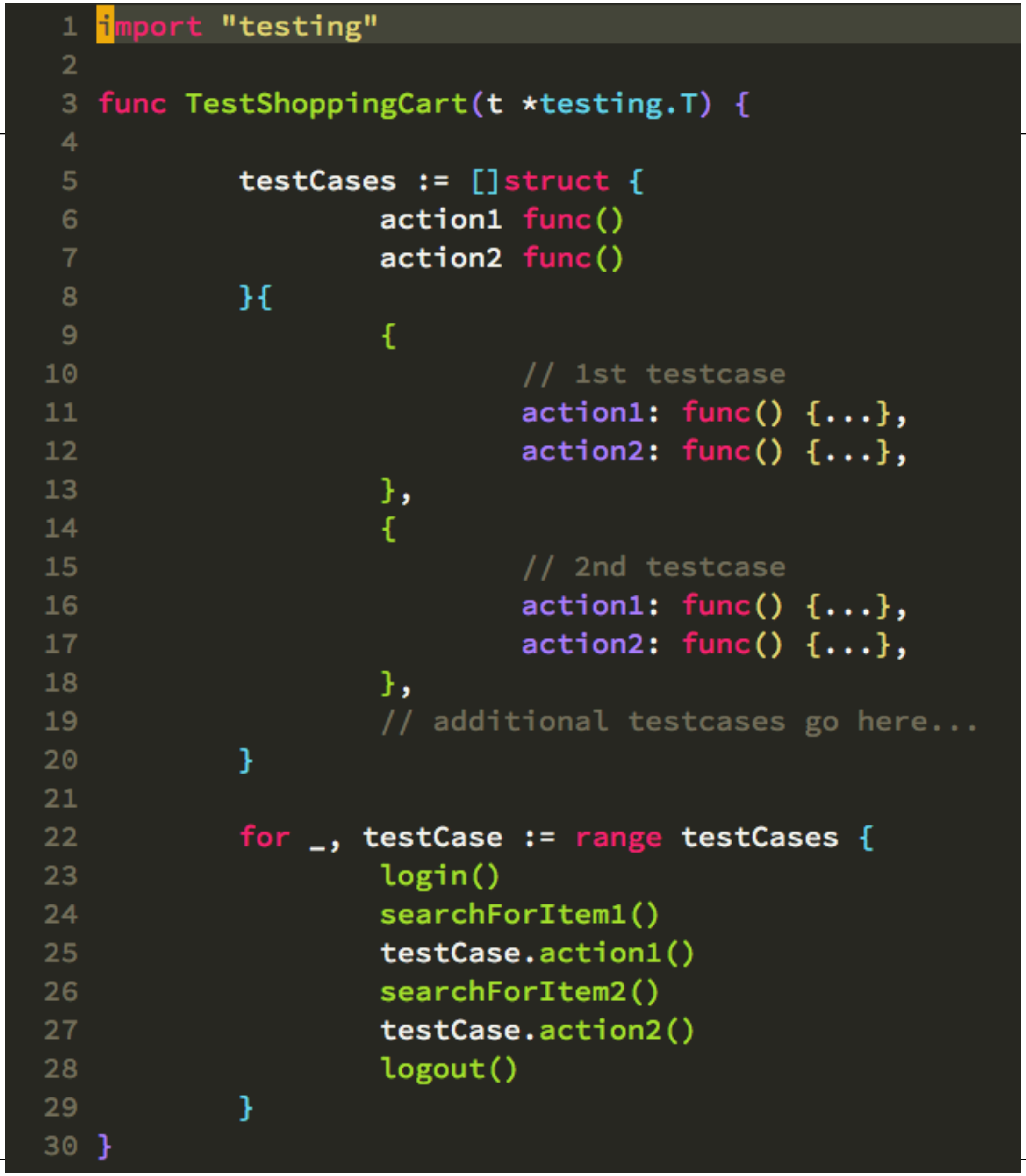
- Template Pattern:



```
public abstract class ShoppingCartTemplate extends TestCase {
    public ShoppingCartTemplate(String pName) { super(pName); }
    protected void setUp() {
        login();
        searchForItem1();
        cartAction1(); // <--- test hook 1
        searchForItem2();
        cartAction2(); //<---- test hook 2
        logout();
    }
    //Subclasses are supposed to implement these test hooks anyway they like.
    abstract protected void cartAction1();
    abstract protected void cartAction2();
    //test tool specific implementations of methods
    //(e.g. login(), searchForItem1(), etc.)
}
```

Existing Best Practices – Systems Level

- Similar idea in Go:
  - Anonymous structs contain test hooks
  - Array of structs as test suite
  - For loop through test cases
- Helps to focus on the actual test logic
- Common setup functionality shared

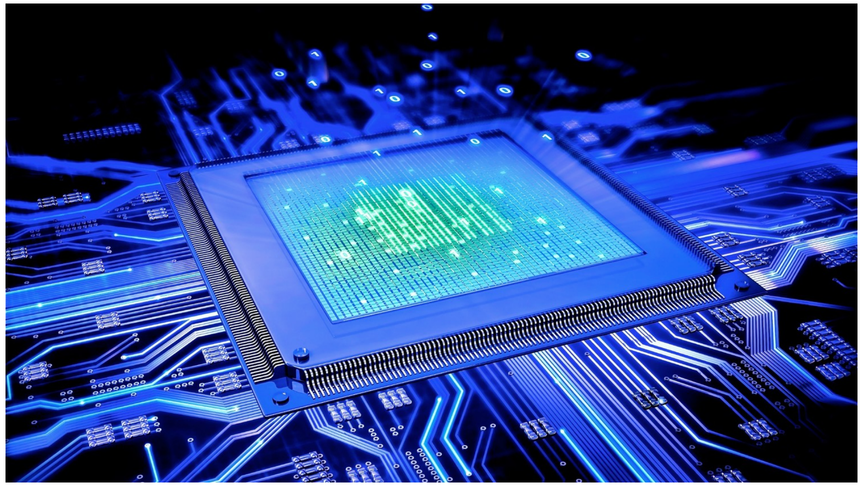


```go
import "testing"

func TestShoppingCart(t *testing.T) {

	testCases := []struct {
		action1 func()
		action2 func()
	}{
		{
			// 1st testcase
			action1: func() {...},
			action2: func() {...},
		},
		{
			// 2nd testcase
			action1: func() {...},
			action2: func() {...},
		},
		// additional testcases go here...
	}

	for _, testCase := range testCases {
		login()
		searchForItem1()
		testCase.action1()
		searchForItem2()
		testCase.action2()
		logout()
	}
}
```

Missing Best Practices

- Mostly embedded systems area, i.e. systems where the access is limited
  - Introduction of automated testing is more complex

- People tend to build specialized test frameworks for more complex Non-functional tests (performance, security)

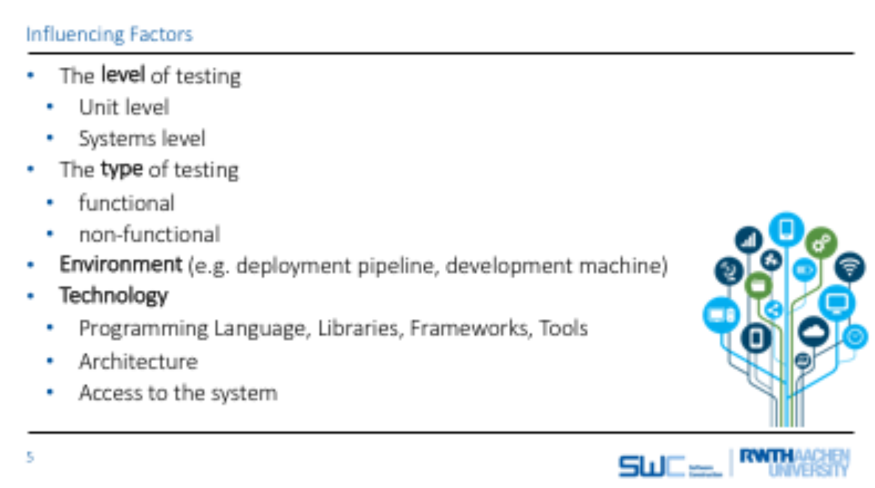


Results

- Lots of resources for implementing tests code-wise
- Rich ecosystem for writing tests in the domain of Web-based systems
- Best practices are missing where foundation is missing
    - i.e. tool that support setup, execution of tests

Summary

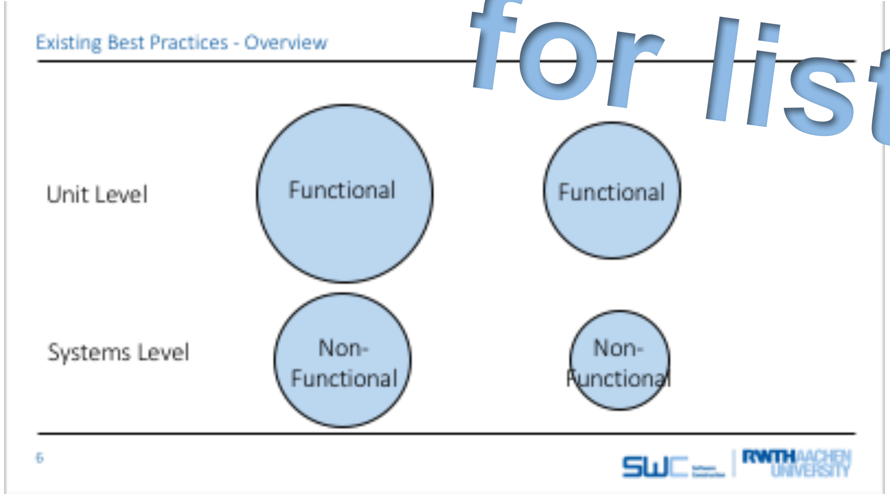
Thanks for listening



Influencing Factors

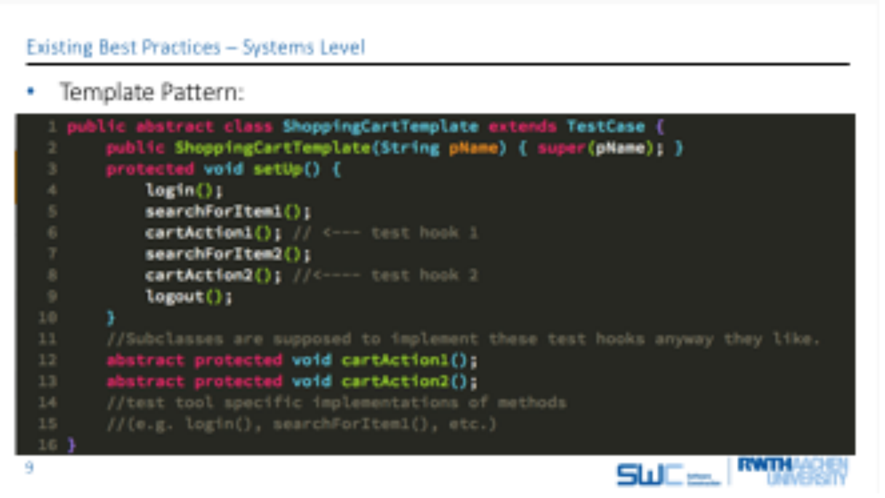
- The **level** of testing
  - Unit level
  - Systems level
- The **type** of testing
  - functional
  - non-functional
- **Environment** (e.g. deployment pipeline, development machine)
- **Technology**
  - Programming Language, Libraries, Frameworks, Tools
  - Architecture
  - Access to the system

5



Existing Best Practices - Overview

6

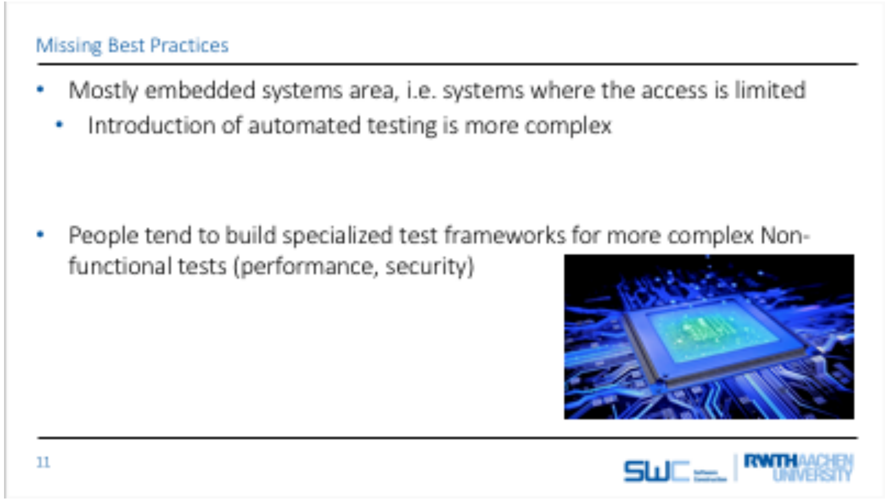


Existing Best Practices – Systems Level

- Template Pattern:

```java
public abstract class ShoppingCartTemplate extends TestCase {
    public ShoppingCartTemplate(String pName) { super(pName); }
    protected void setUp() {
        login();
        searchForItem1();
        cartAction1(); // <--- test hook 1
        searchForItem2();
        cartAction2(); //<---- test hook 2
        logout();
    }
    //Subclasses are supposed to implement these test hooks anyway they like.
    abstract protected void cartAction1();
    abstract protected void cartAction2();
    //test tool specific implementations of methods
    //(e.g. login(), searchForItem1(), etc.)
}
```

9

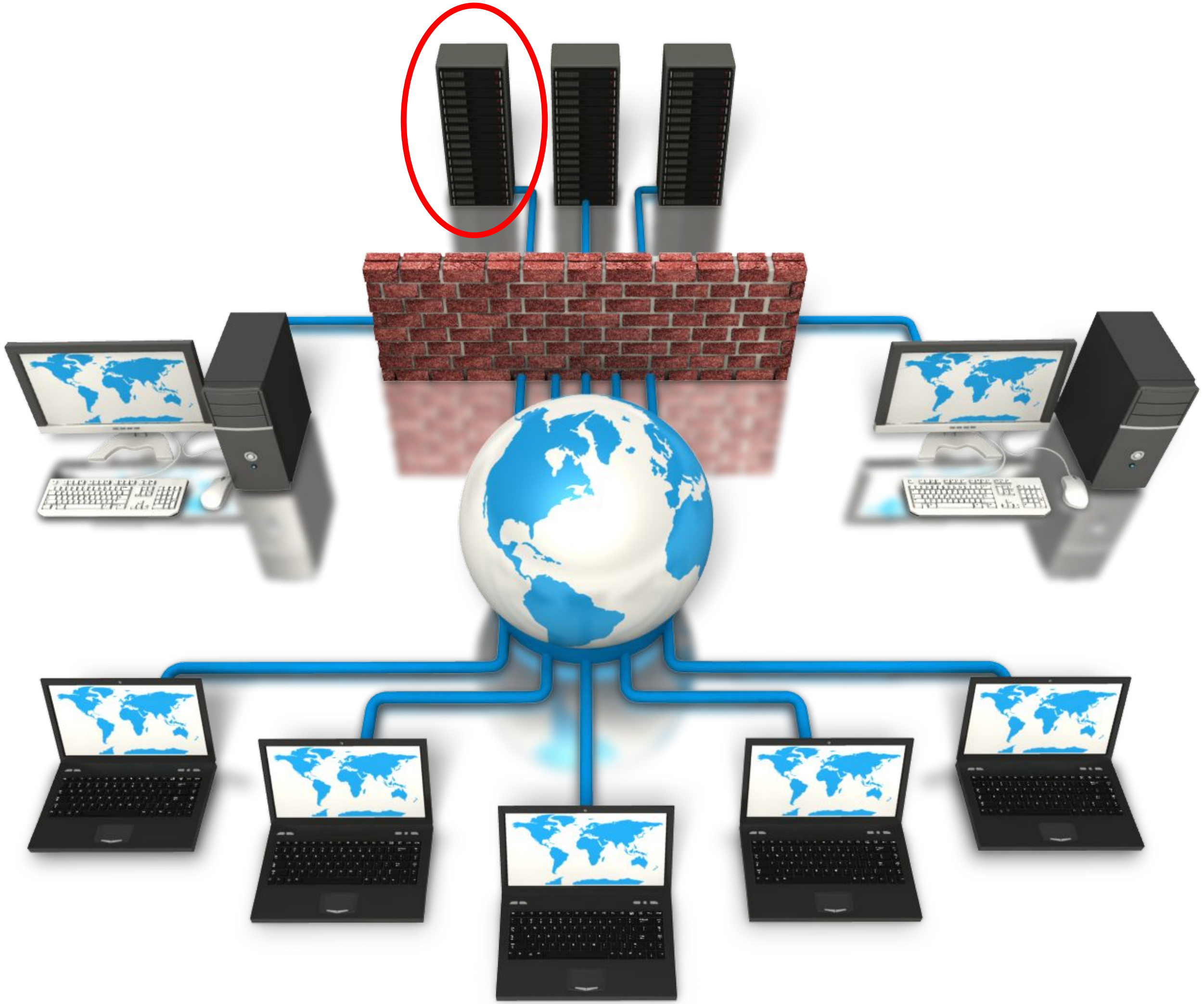


Missing Best Practices

- Mostly embedded systems area, i.e. systems where the access is limited
  - Introduction of automated testing is more complex
- People tend to build specialized test frameworks for more complex Non-functional tests (performance, security)

11

Why bother testing Infrastructure?

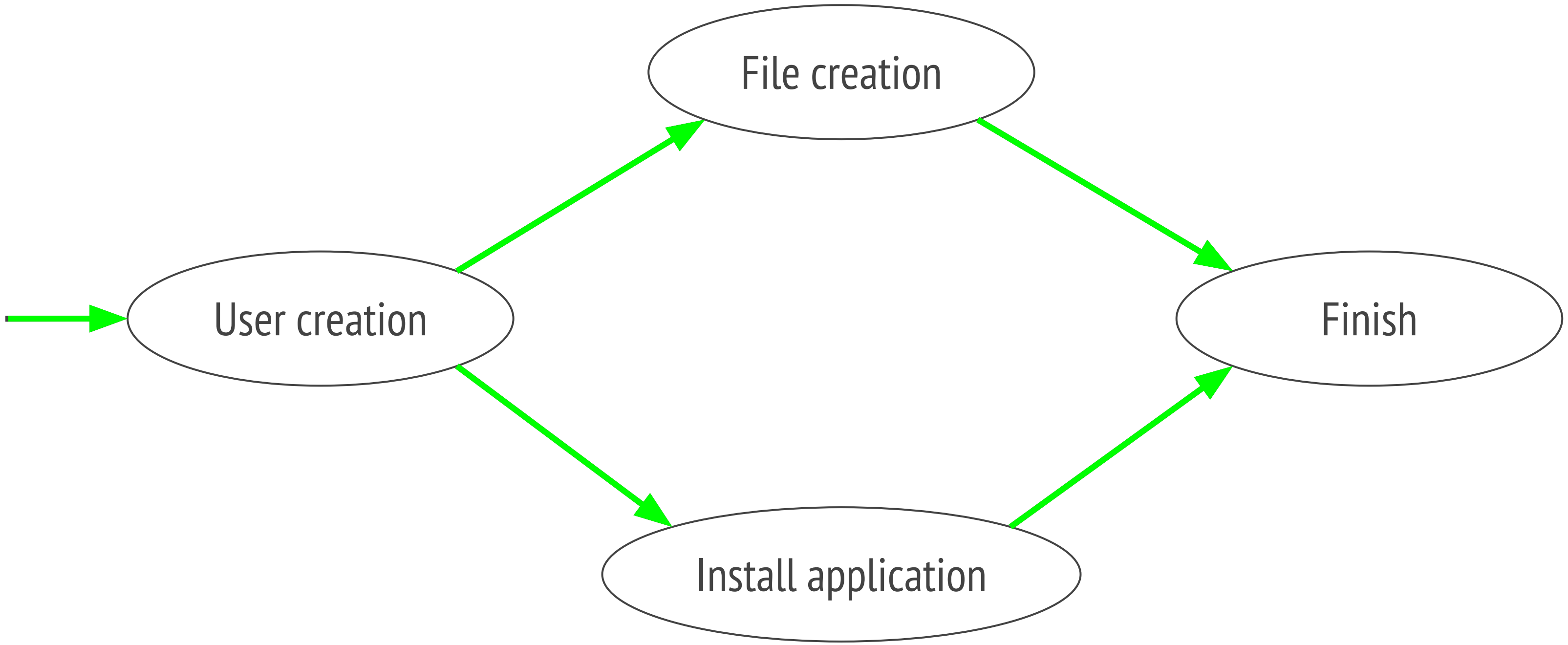


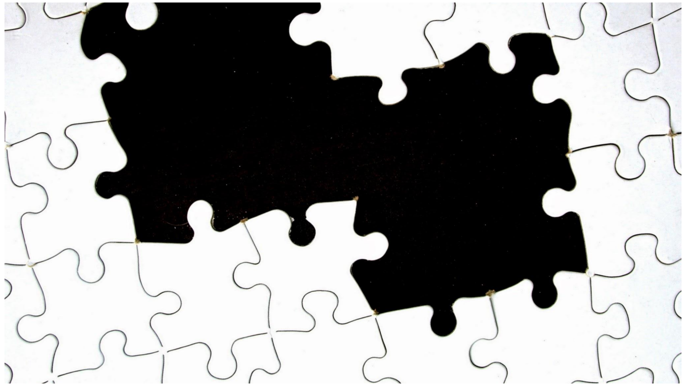
```yaml
- name: Create directory
  file:
    path: /opt/some_directory
    state: directory
```

“Treat the configuration of systems the same way that software source code is treated.”

Kief Morris
(author of “Infrastructure as Code”)

Functional testing	✓
Conformance testing	✓
Idempotence testing	?
State-based testing	?


File creation
User creation
Finish
Install application



THANKS!

2017, Feb 09, Marco Moscher

marco.moscher@rwth-aachen.de

Facing Synthetic Workload Generation as part of Performance Testing – a tools approach

Seminar - The Art of Software Testing

Supervisor - Konrad Fögen, foegen@rwth-Aachen.de

Performance Testing

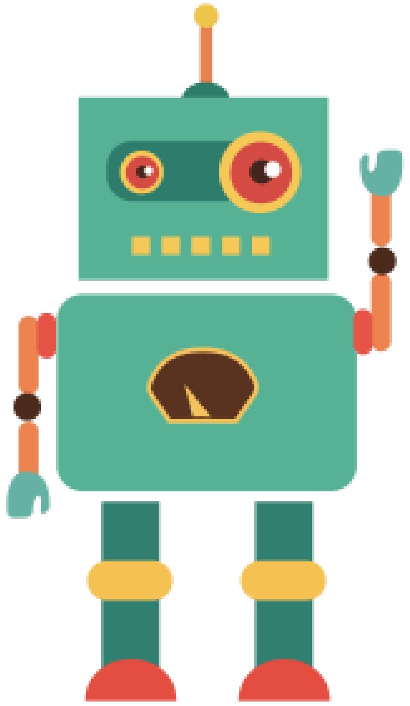
Performance Testing (recap)

- Highly specialized task
- Practice during software testing
- Clinches directly the customer satisfaction
- Evaluate responsiveness, stability & resource utilization under load

Study

State of the Art

- Research on Performance Testing conducts ..
  - .. simplification of test definition
  - .. generalization of definitions (DSLs)
  - .. automation of artifact generation
  - .. adaptability of various approaches
- But .. what's about Runtime Data?



400

Bad Request

Challenge / Research

- Performance Tests without possessing any (runtime-) data
- Research Question ..

  … is synthetic workload deduction considered
- Presumption: no base- nor runtime data yet exists

Bidder SUT (case study)

- e-commerce platform yet in development
- Early considerable performance testing ...
  - .. simulate 100k unique users
  - .. test three different user roles

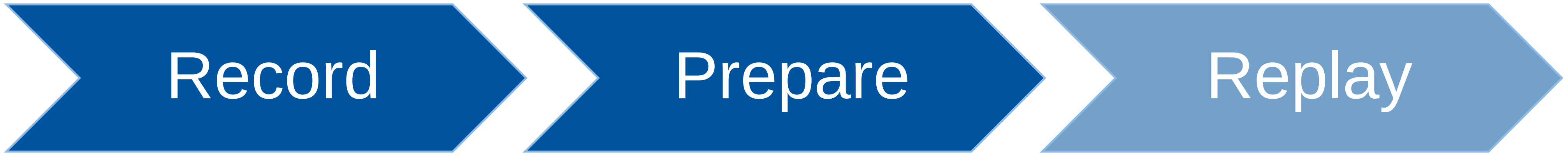
Existing Approaches

Existing Approaches

- two approaches widely used
    - Capture and Replay (CR)
    - Model-Based-Testing (MBT)
- Generating test-scripts, test-artifacts and workload
- Focusing workload generation (RQ + Presumption)

Capture & Replay

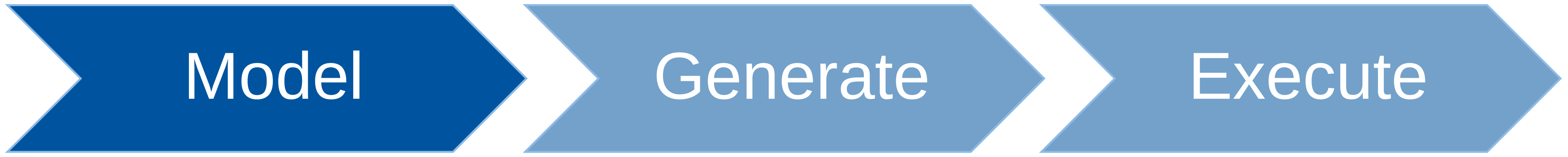
- Example: Microsoft Visual Studio



- Step Prepare requires external data source (contradiction to presumption)

Model-Based-Testing

- Example: Canopus



- Step Model requires external data source (contradiction to presumption)

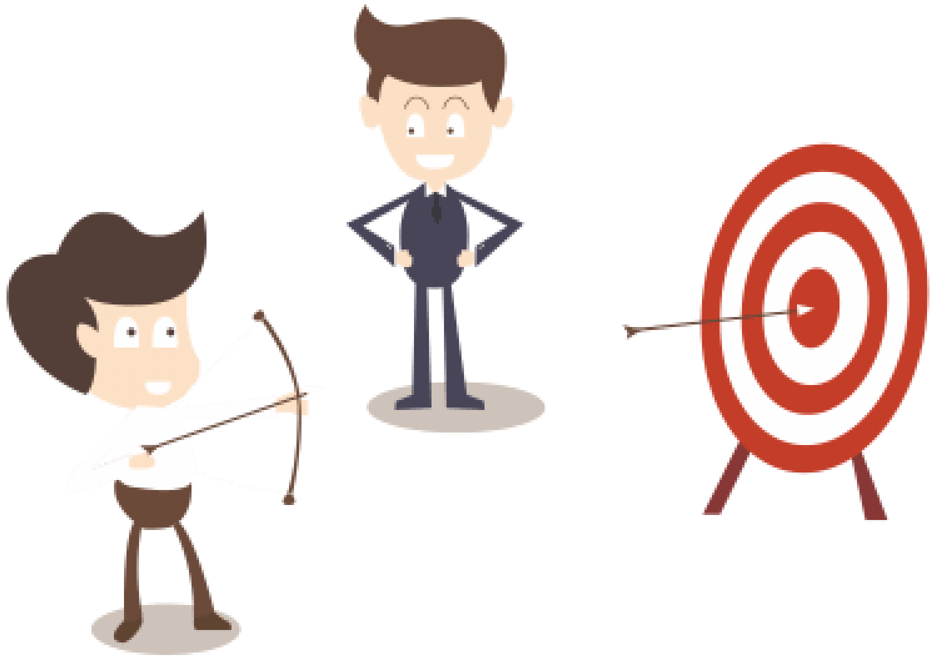
Approaches Insufficient

- Data generation not addressed
  - Problem: workload does not mean runtime data

- Performance Tests mostly after first deployment
  - not during development

How to solve this Problem?

- Focus external data generation
- Obtain synthetic workload which meets ..
    - .. uniqueness?
    - .. causal relations?

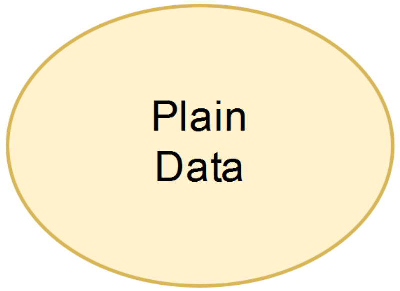


Obtain Synthetic Workload

Obtain Synthetic Workload

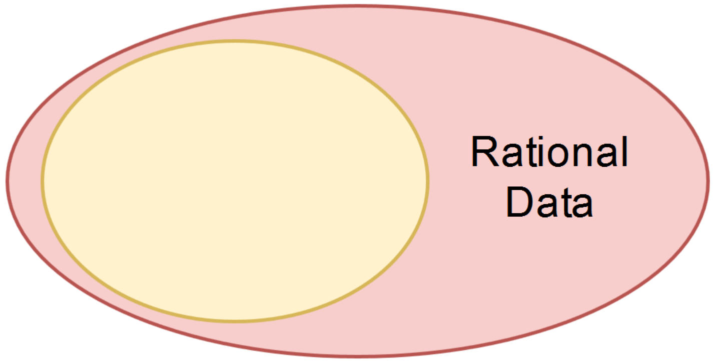
- Many tools exists
- Combine tools regarding data complexity ..
  - .. use equivalence classes
  - .. Plain Data ≤ Rational Data ≤ Complex Data
- Evaluate if required data can be derivated within these classes

Plain Data



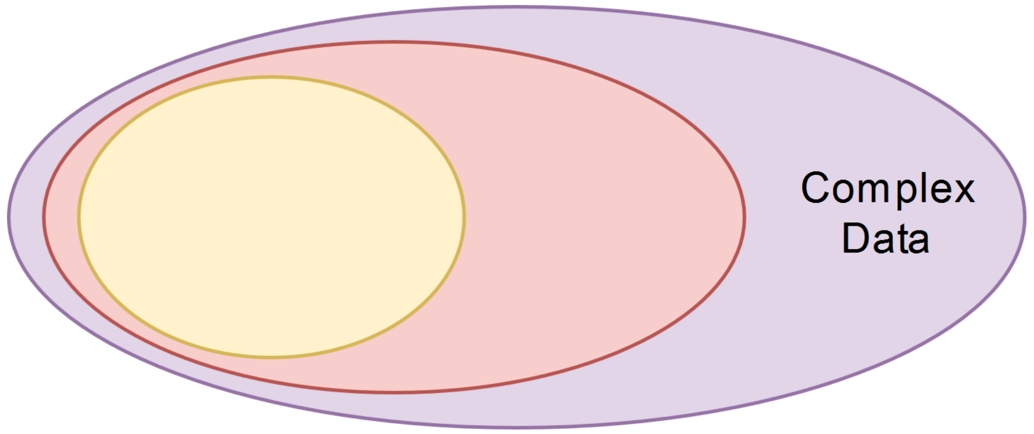
- Single, unrelated data entities
- Example - User Accounts (Name, Address, Email etc.)
- Tools - faker.js

Rational Data



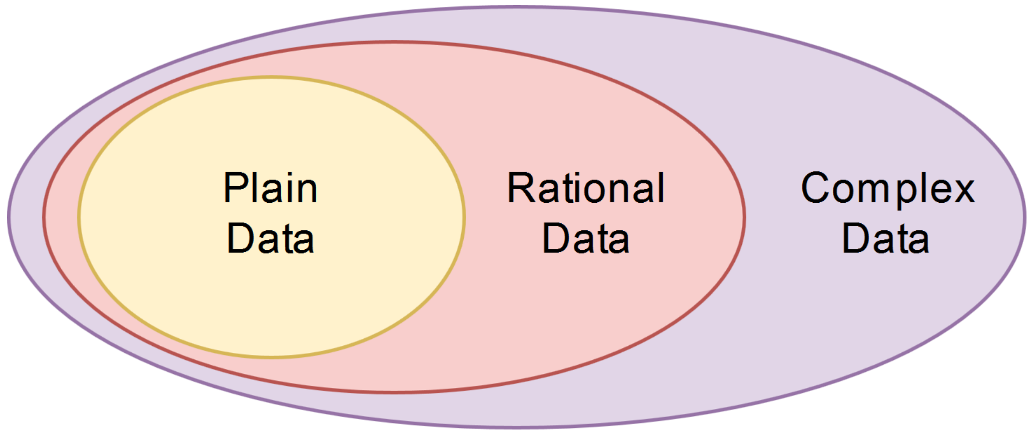
- Related data, i.e. database constraints
- Example - 5 Products for each Users
- Tools - redgate sql, benerator

Complex Data



- Ordered data
  - $P(b_i) = price\ of\ bid$
  - $P(b_i) < P(b_{i+1})$
- Example - 5 bids on one product of different users
- Tools - redgate sql, benerator

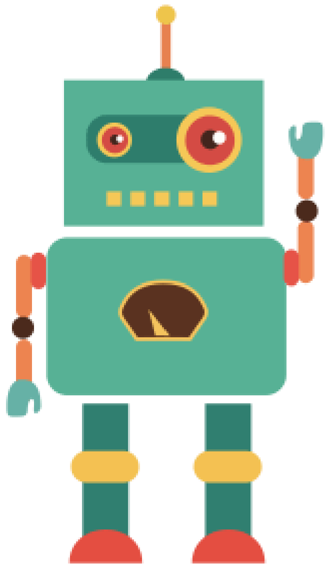
Summary



- Classes sufficient enough (w.r.t Bidder)
- Tool support on every complexity-level exists
- External Data sources creatable using existing tools

Conclusion

Synthetic Workload Generation



No Hot Research Topic

Present Approaches Insufficient

Tool Combination Needed

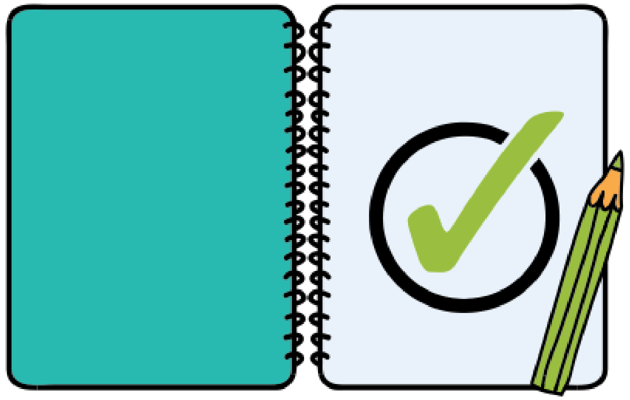
Outlook

- Extend MBT Approach with additional data-related meta-models
- Introduce Interconnection ..
  - .. DSL
  - .. easier Import / Export

Results

Thanks!

- Time consuming when dealing with missing test data
- Synthetic workload generation is a complicated task
- Tools are not yet connectable



Credits / Resource Links

- Slide 2 „Performance Testing“
    - Notebook Icon - https://testinsane.com/performance-testing-services.html

- Slide 5 „State of Art“, Slide 24 „Synthetic Workload Generation“
    - Robot Icon - https://testinsane.com/automation-testing-services.html

- Slide 6 „Gateway not found“
    - Cat Image - https://http.cat/

- Slide 8 „Bidder SUT“
    - System Under Test Icon - https://testinsane.com/services.html

- Slide 10 „Existing Approaches“
    - Notebook Icon - https://testinsane.com/functional-testing-services.html

- Slide 16 „Data Derivation“, Slide 24 „Synthetic Workload Generation“
    - Working Human - https://testinsane.com/work-culture.html

- Slide 21 „Results“
    - Check Icon - https://testinsane.com/compliance-testing-services.html

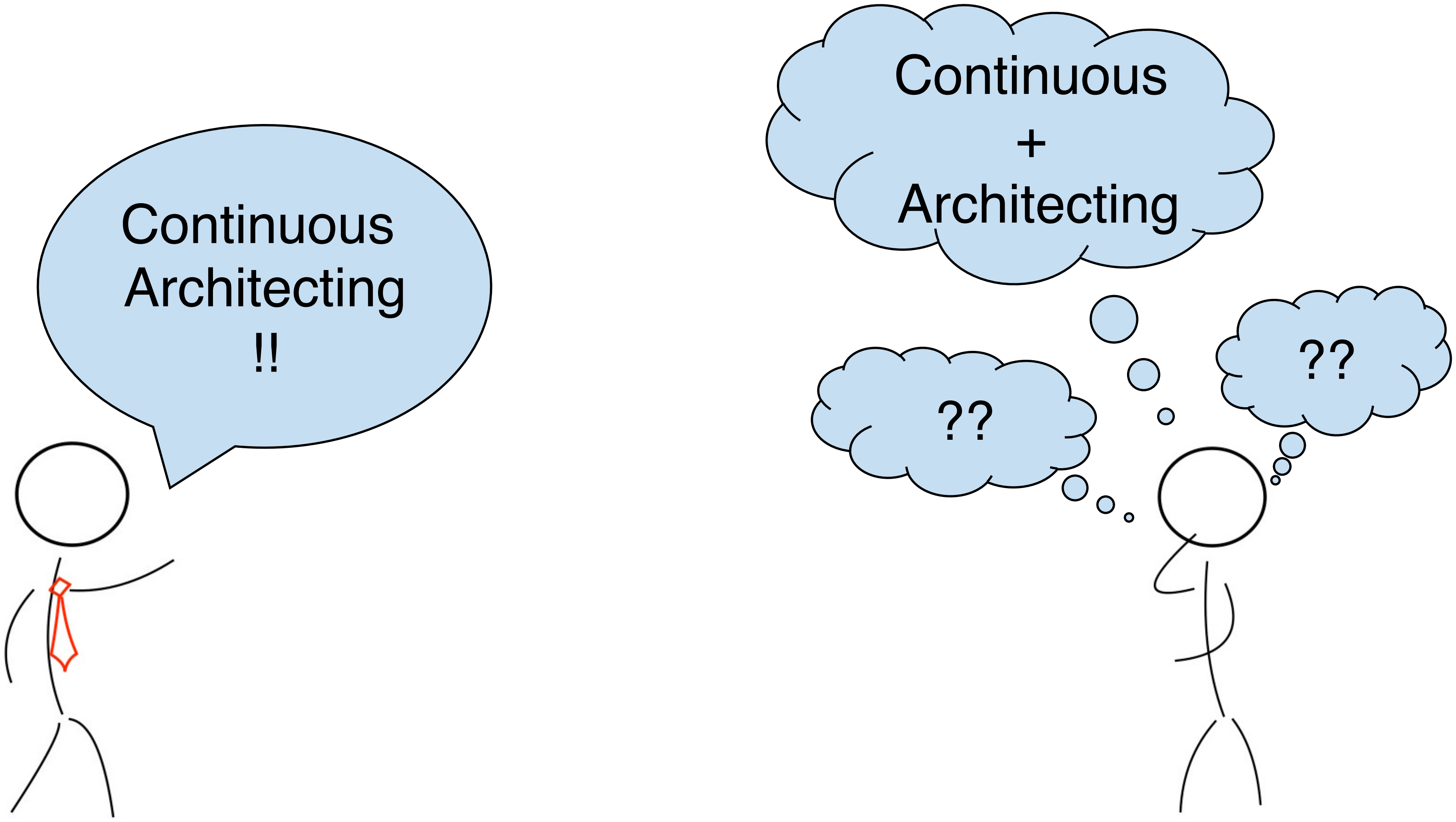
- Slide 23 „Outlook“, Slide 24 „Synthetic Workload Generation“
    - Person Icon - https://testinsane.com/ui-ux-testing-services.html

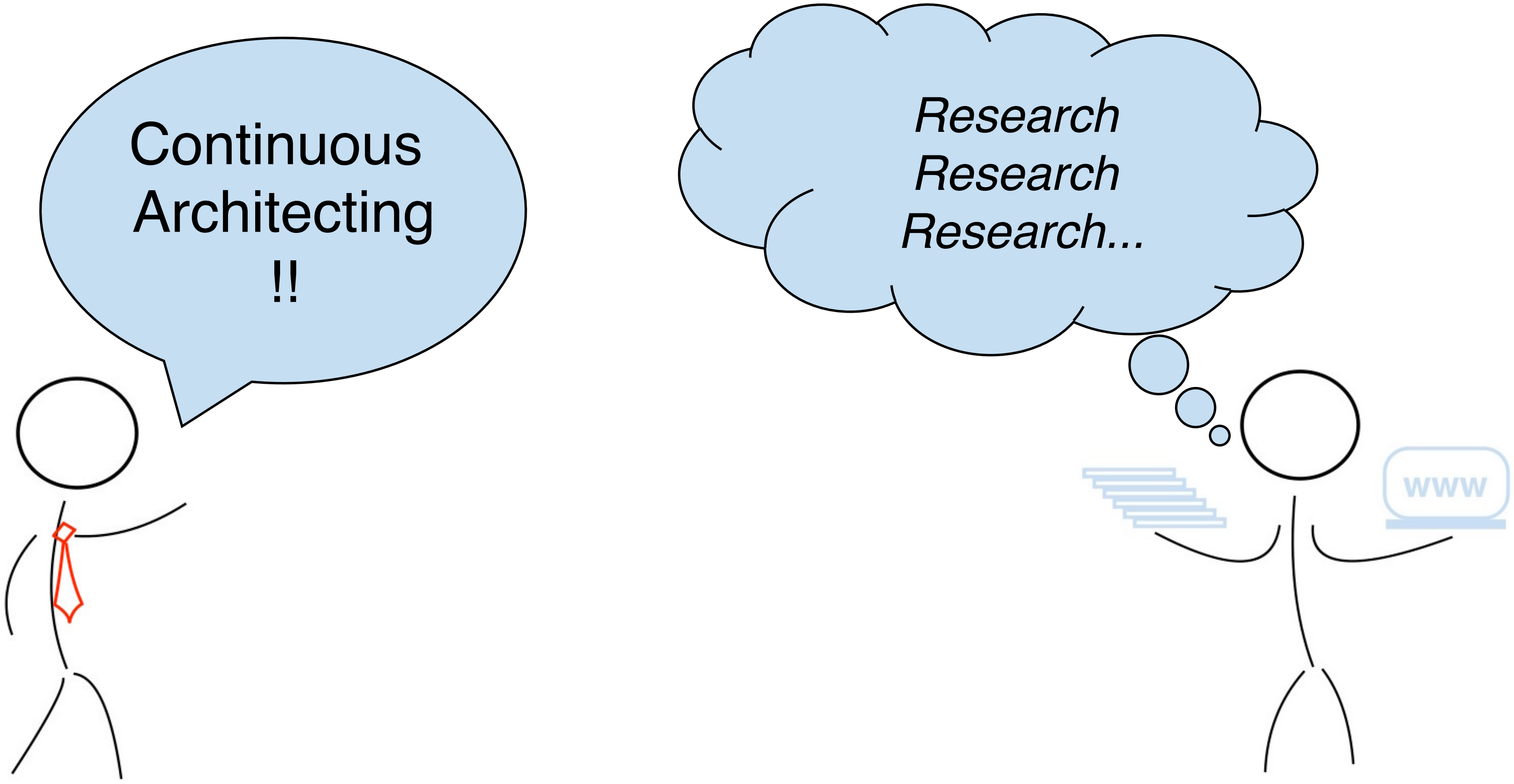
- Overall Fonts
    - Headline – Google, Astigmatic, Lukiest Guy - https://fonts.google.com/specimen/Luckiest+Guy
    - Text, Highlights – Google, Vernon Adams, Oxygen - https://fonts.google.com/specimen/Oxygen

Benedikt Holmes,
benedikt.holmes@rwth-aachen.de

Continuous Architecting

Just another buzzword?


Continuous Architecting !!
Continuous + Architecting
??
??


Continuous Architecting !!
Research Research Research...
www

Perception A

- Big-data domain
- Aid architecture design and maintaince
- Technical analysis framework (OSTIA)

Solved Problems:

- Architecture erosion

Perception B

- Mostly embedded software domain
- Architecture refactoring tasks are often low prioritized
- Organisational framework (CAFFEA)

Solved Problems:

- Architecture erosion
- Communication issues / Lack of communication

Perception C

- Bottleneck in software development life cycle
- Architecture style to support agile delivery methods

Solved Problems:

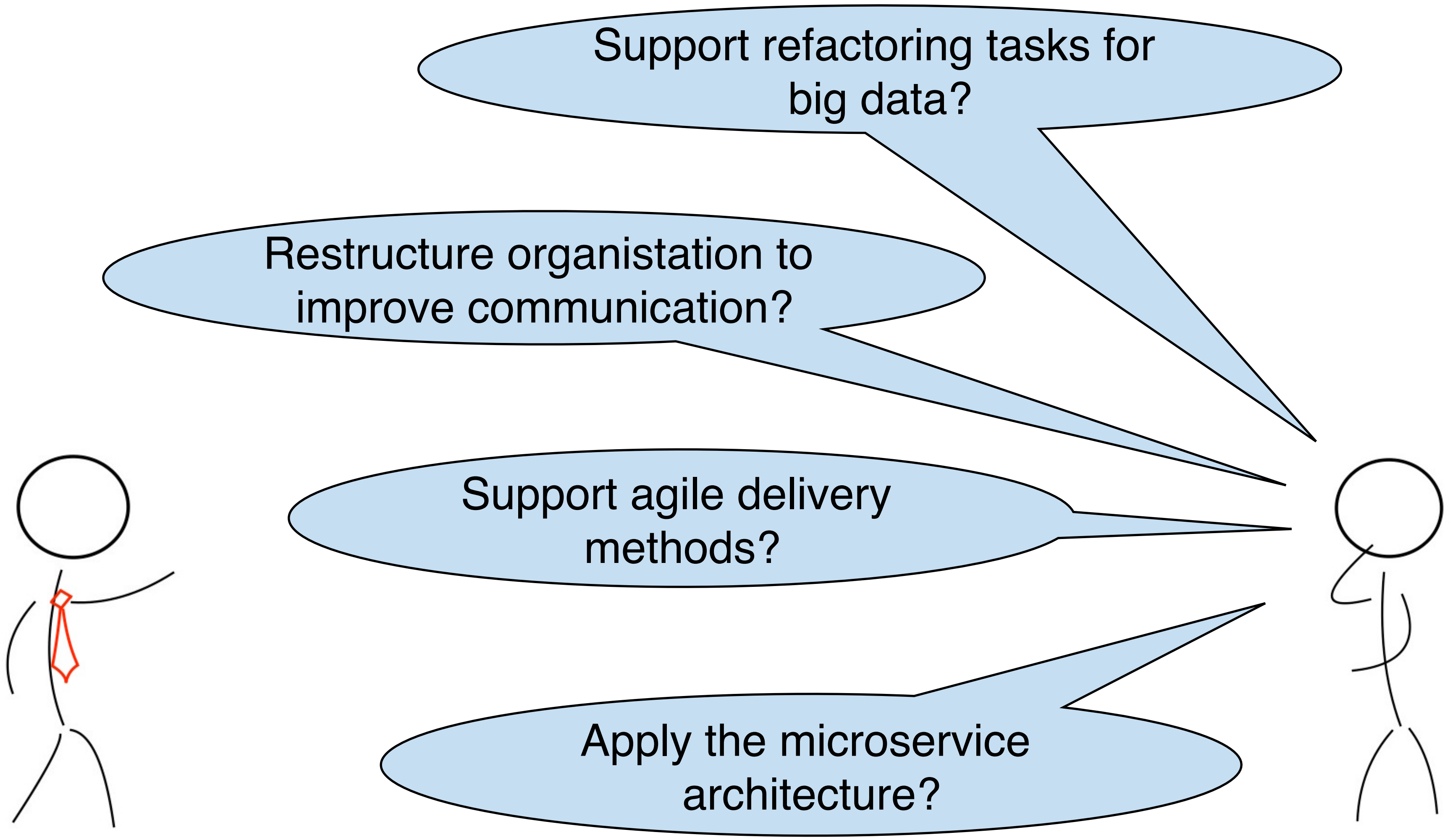
- Architecture erosion
- Slow response to changing requirements

Perception D

- Micro-service architecture

Solved Problems:

- Architecture erosion
- Communication issues


Support refactoring tasks for big data?
Restructure organisation to improve communication?
Support agile delivery methods?
Apply the microservice architecture?

!

Be alert

Perception A

Support for big-data architecture refactoring tasks

Perception B

Support architecture refactoring in generall and improve communiation

Perception C

Support agile delivery methods

Perception D

Get rid of architecture erosion and support delivery acceleration

Perception A

Support for big-data architecture refactoring tasks

Perception B

Support architecture refactoring in generall and improve communiation

Perception C

Support agile delivery methods

Perception D

Get rid of architecture erosion and support delivery acceleration

Perception A

Support for big-data architecture refactoring tasks

Perception B

Support architecture refactoring in generall and improve communiation

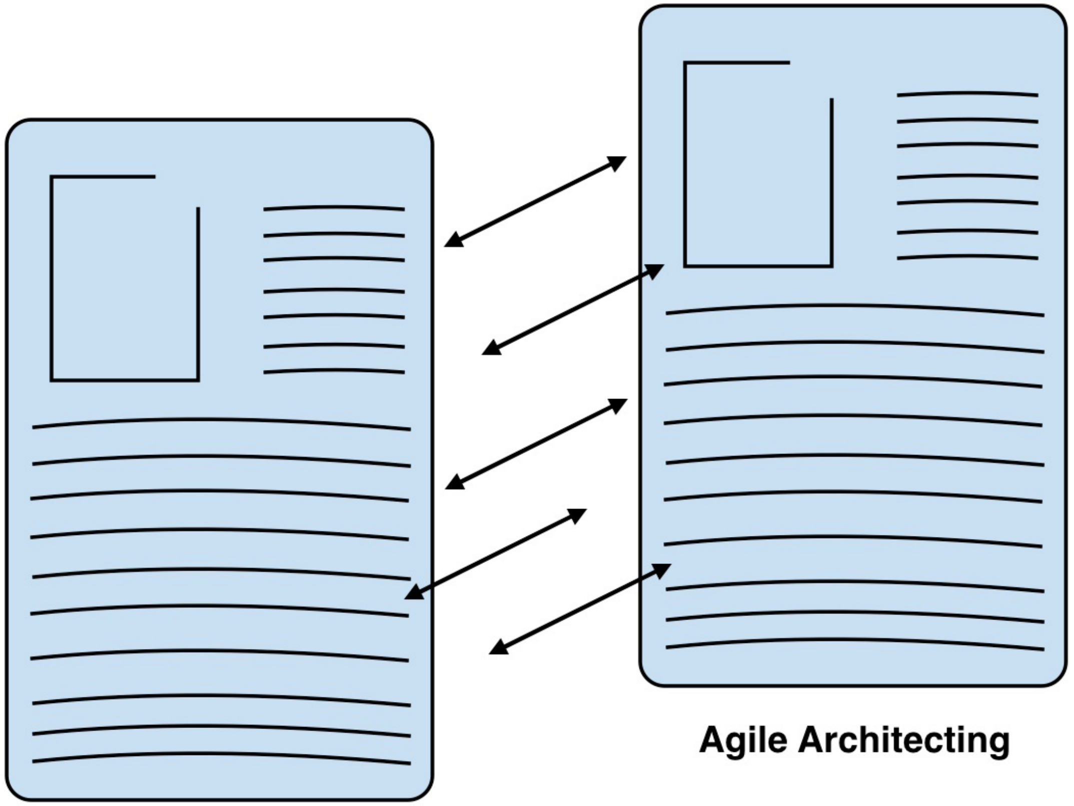
Perception C

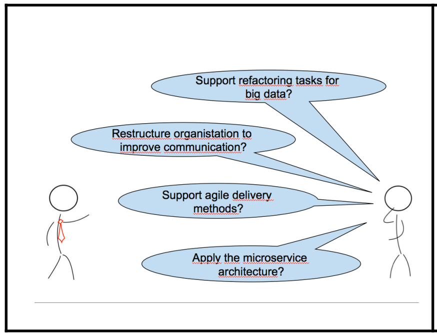
Support agile delivery methods

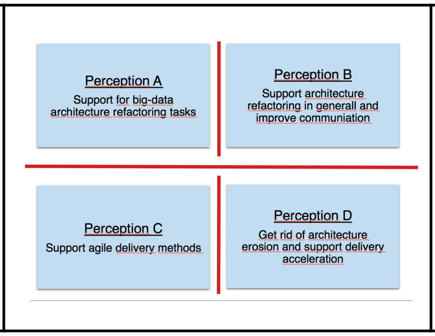
Perception D

Get rid of architecture erosion and support delivery acceleration

1. There is no history of this concept
2. Some parts already exist




Support refactoring tasks for big data?
Restructure organisation to improve communication?
Support agile delivery methods?
Apply the microservice architecture?

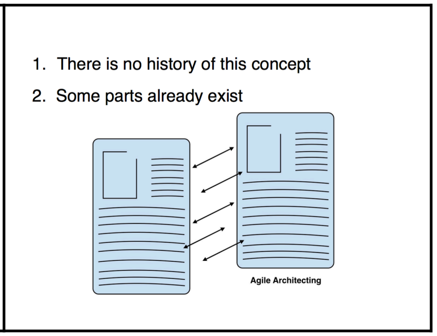


Perception A
Support for big-data architecture refactoring tasks

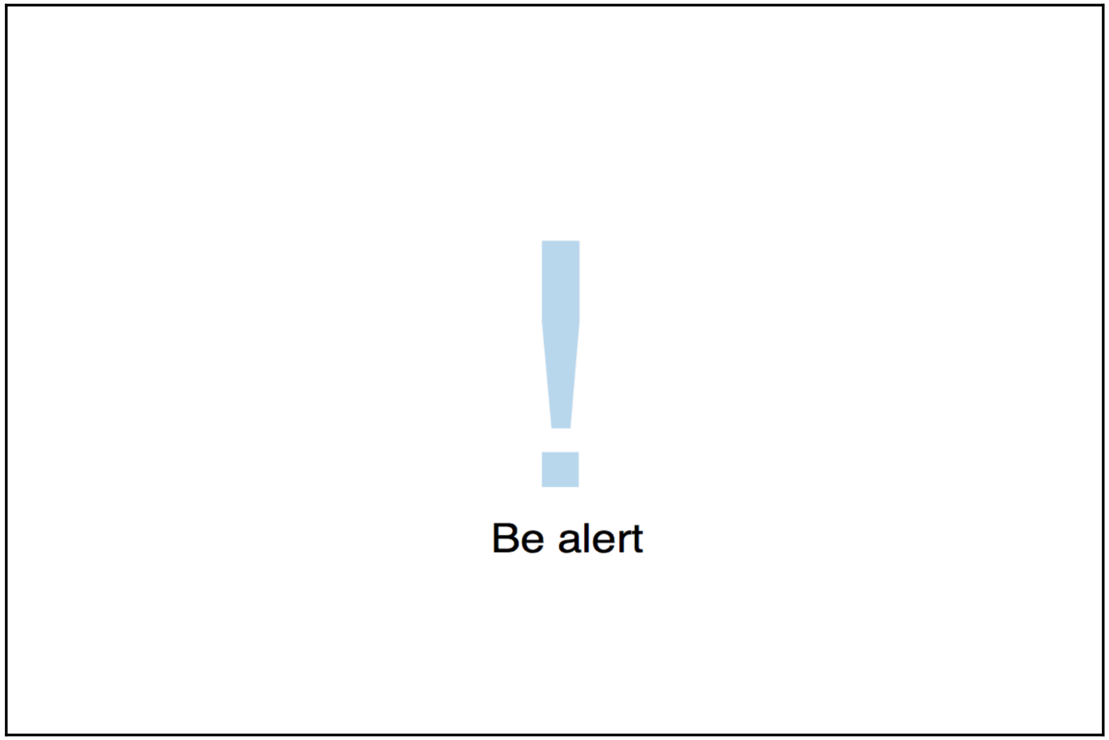
Perception B
Support architecture refactoring in generall and improve communiation

Perception C
Support agile delivery methods

Perception D
Get rid of architecture erosion and support delivery acceleration



1. There is no history of this concept
2. Some parts already exist

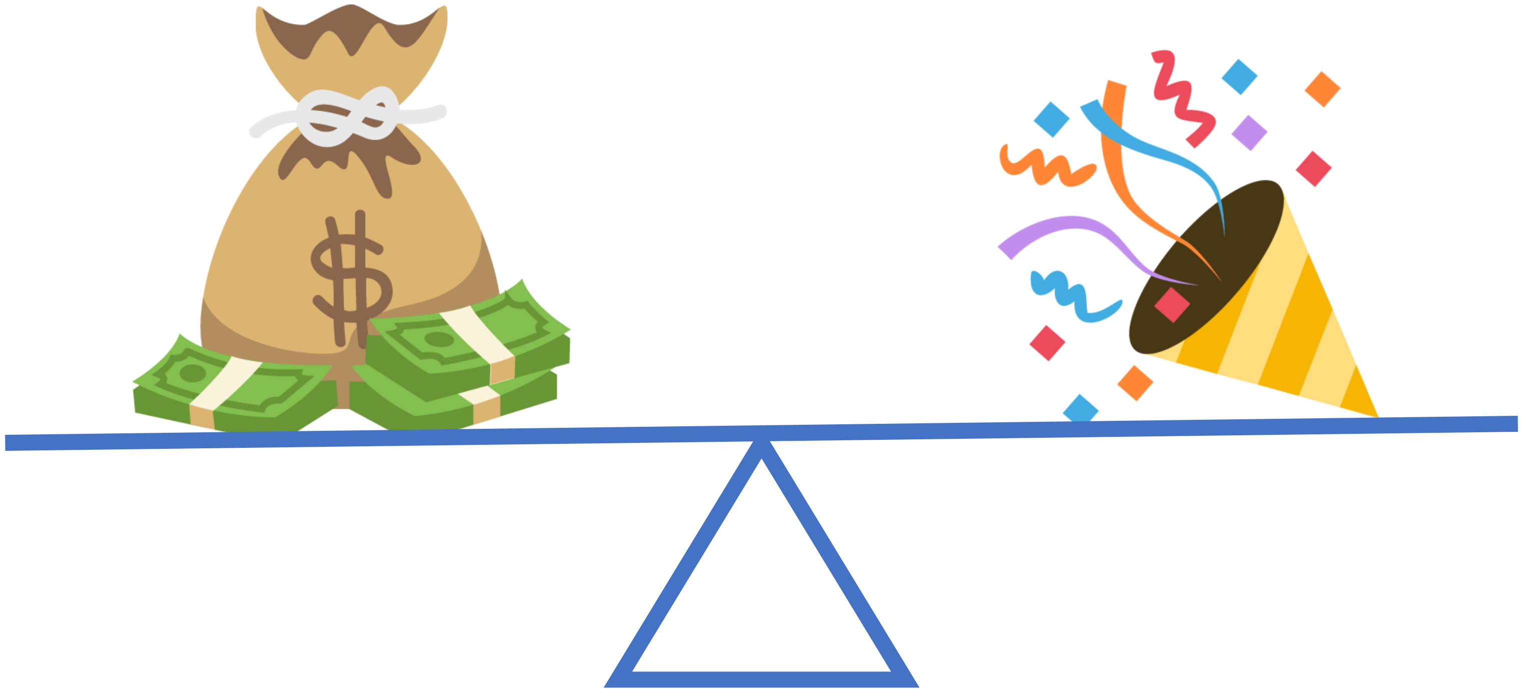


Be alert

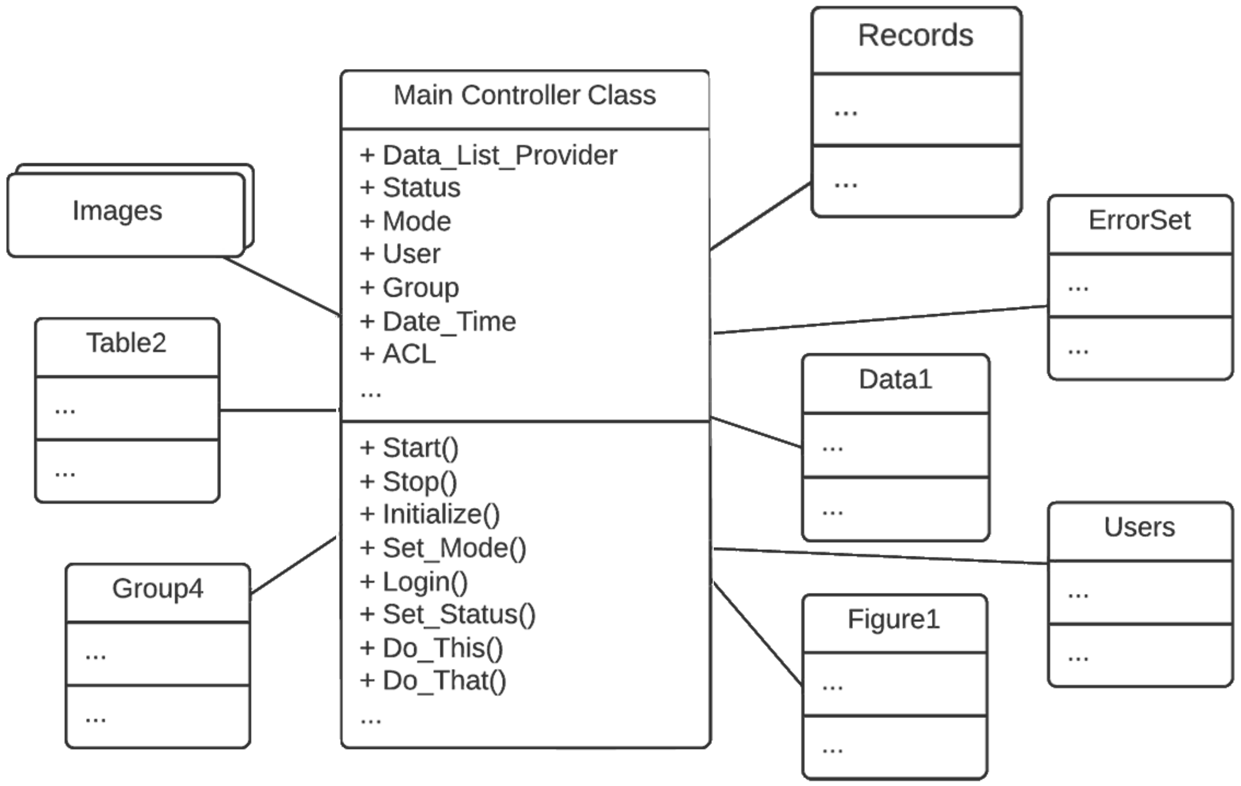
should I Pay or should I Go?

Manage Technical Debt with Cost/Benefit Analysis.

-Radu Coanda-

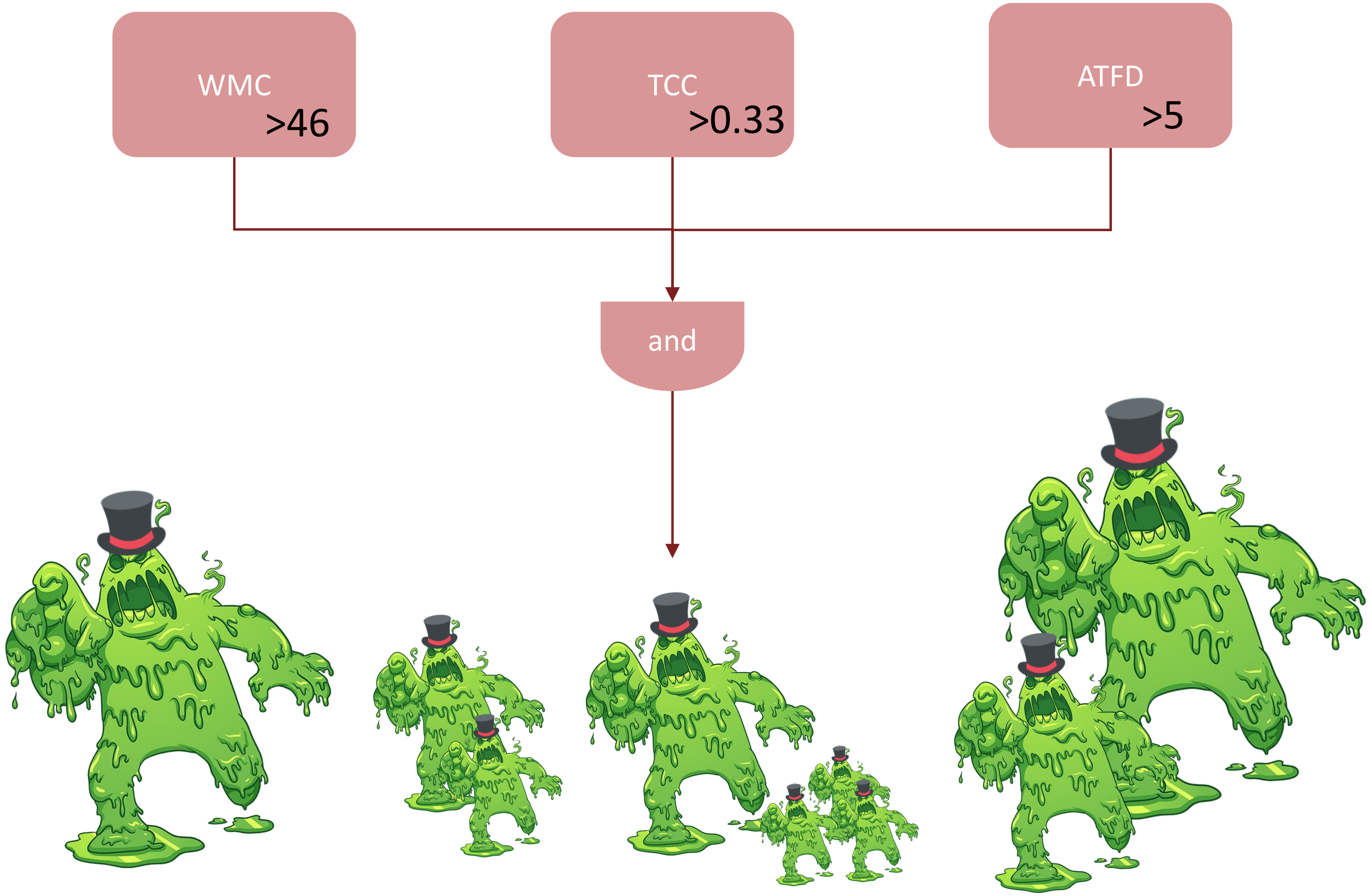


1. Identify appropriate metrics
2. Choose quality characteristics
3. Put everything together in a model



choosing a debt item to pay

choosing a debt item to pay



choose appropriate metrics

choosing appropriate quality characteristics

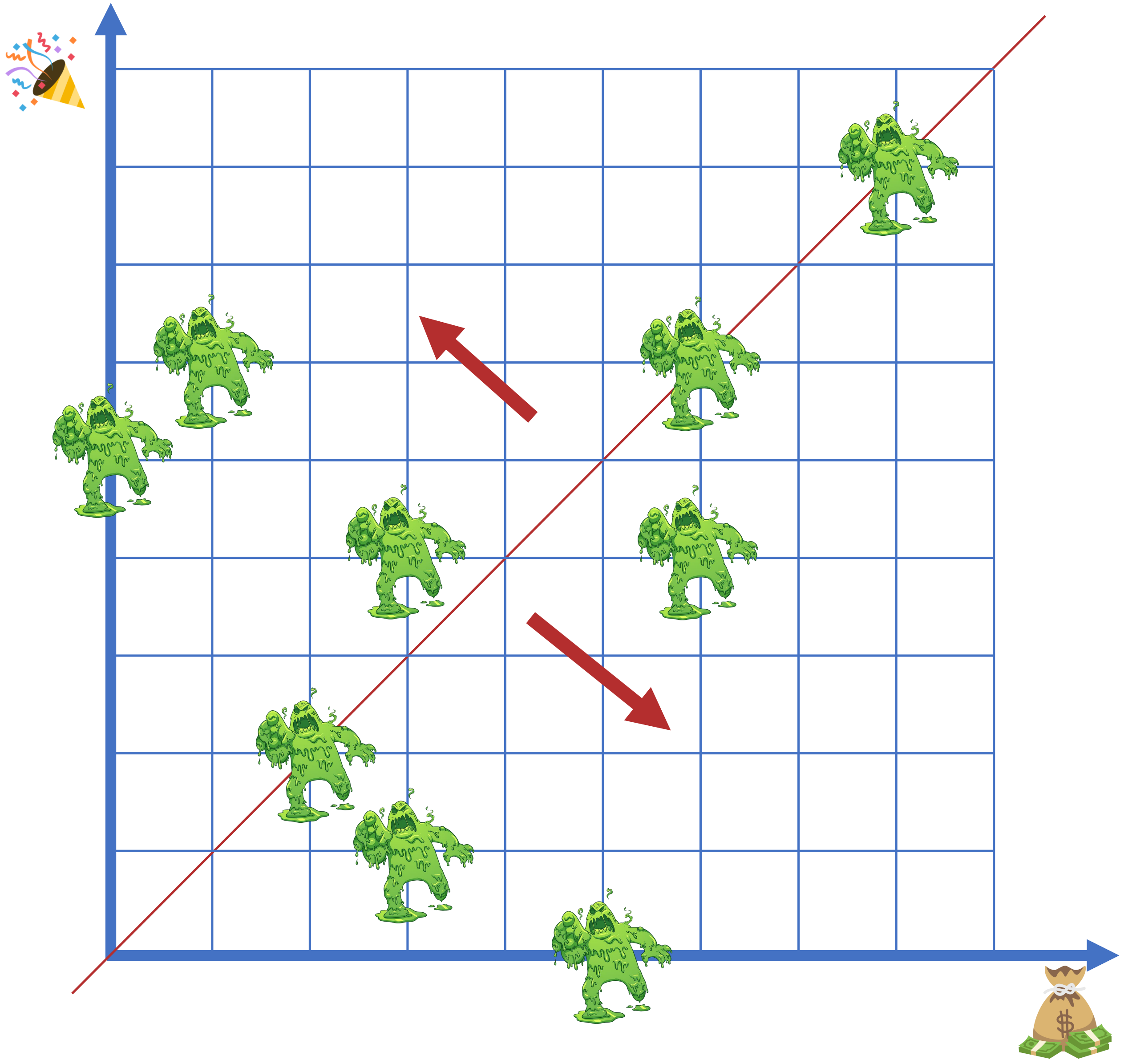
Cost

Blob name	Overall Rank	
	Value	Rank
Blob 1	17	6
Blob 2	22	7
Blob 3	24	9
Blob 4	22	7
Blob 5	11	3
Blob 6	13	4
Blob 7	3	1
Blob 8	5	2
Blob 9	13	4

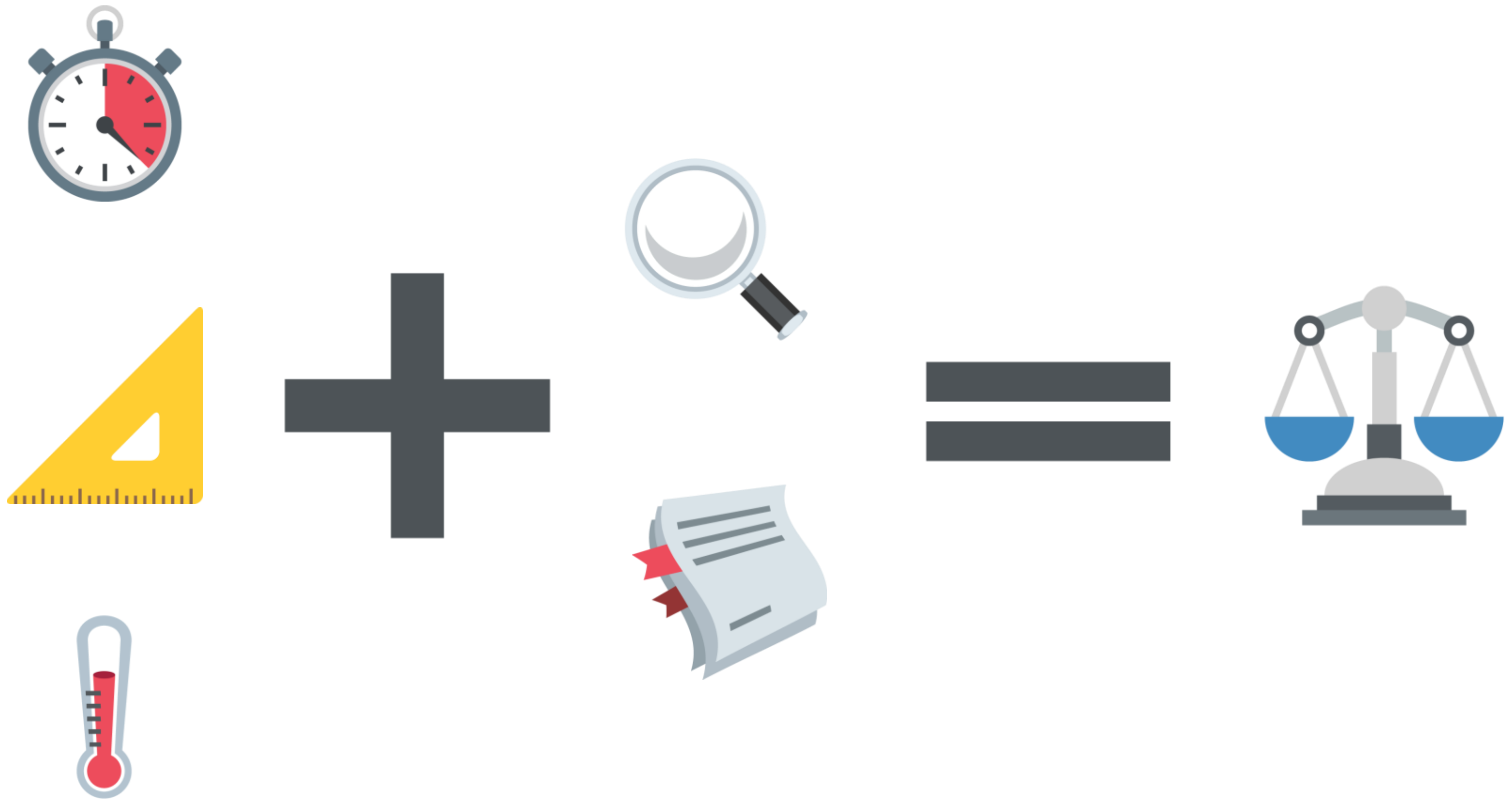
Benefits

Blob Name	Overall Rank	
	Value	Rank
Blob 1	2	1
Blob 2	9	4
Blob 3	14	9
Blob 4	13	7
Blob 5	4	3
Blob 6	11	5
Blob 7	12	6
Blob 8	13	7
Blob 9	3	2

put everything together



put everything together



Don‘t be a Dodo!

The Story of the Dodo

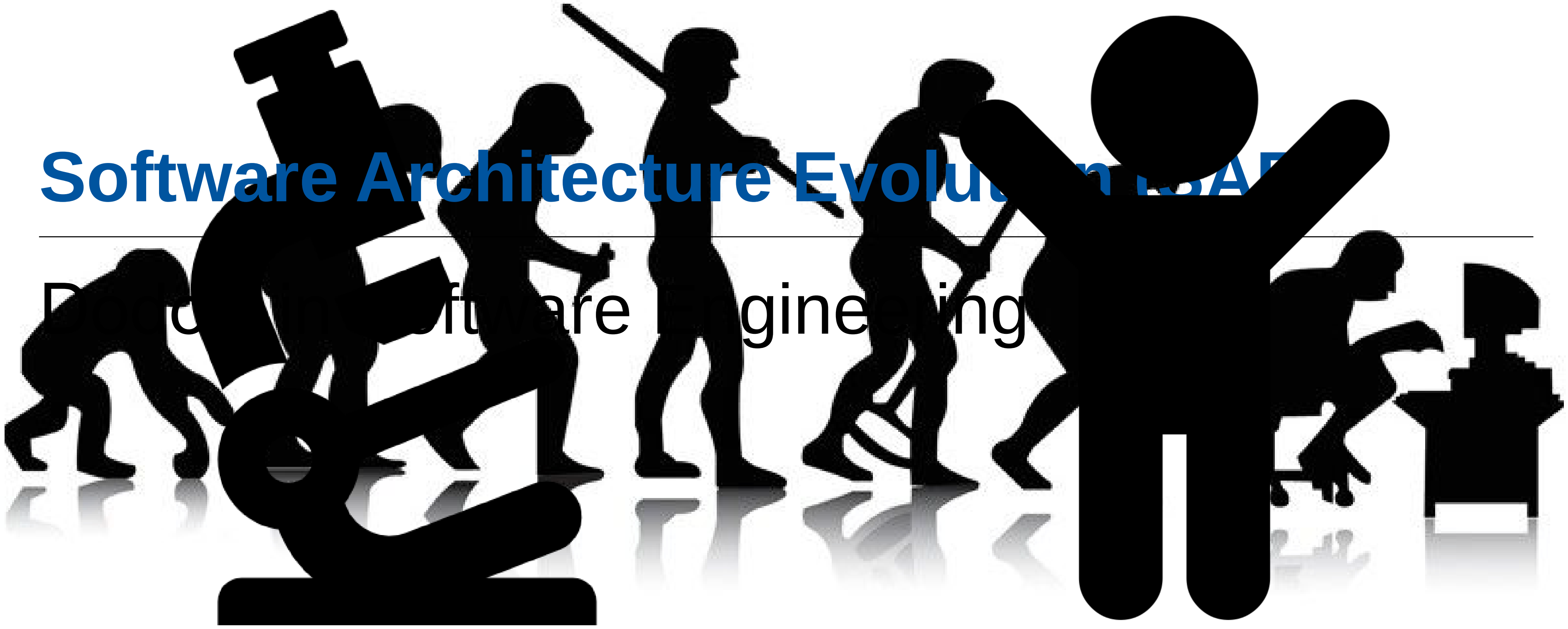
+

Mauritius

=

?

What is it?



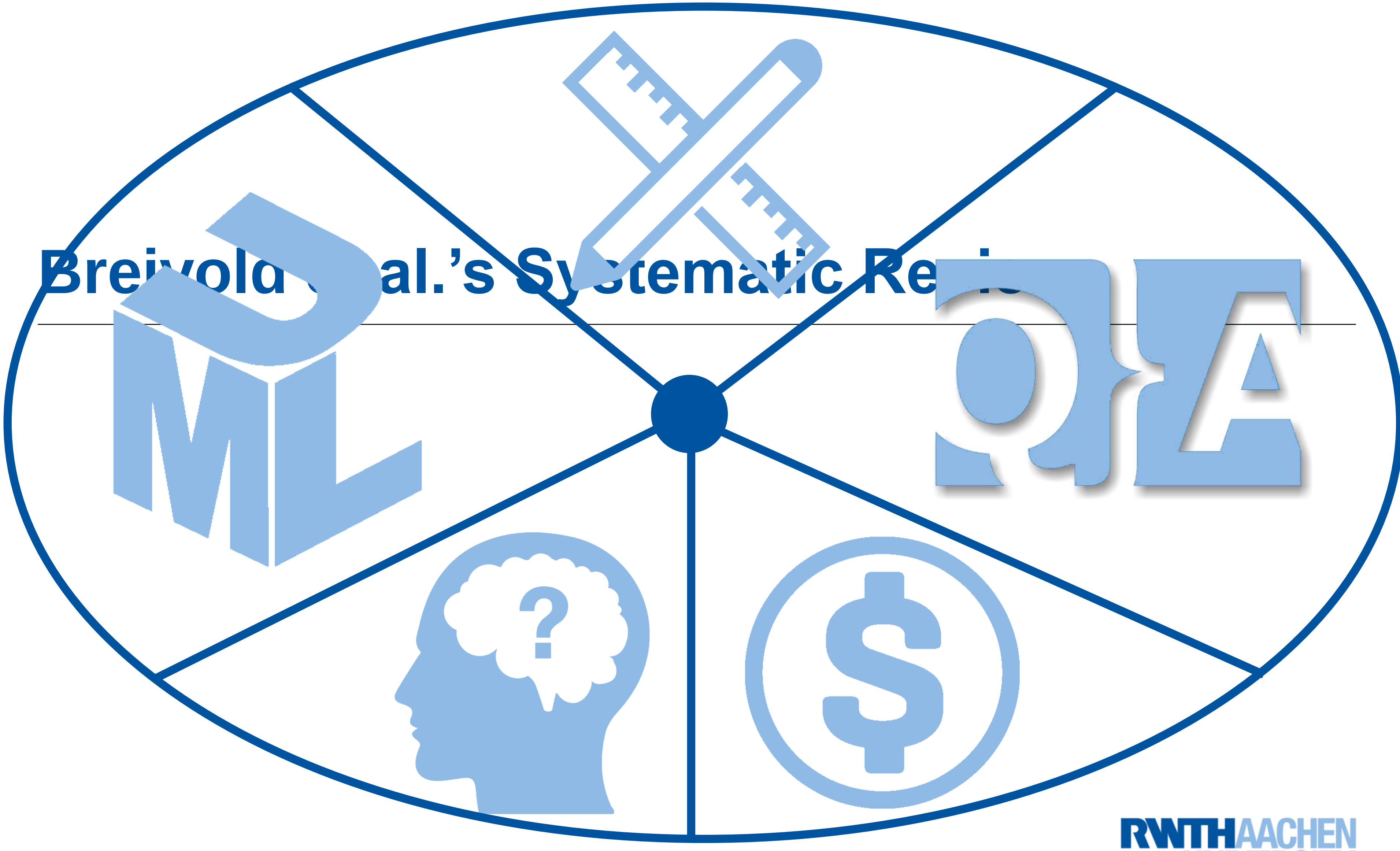
Software Architecture Evolution (SAE)

Doctor in Software Engineering

20 years in 20 minutes

How to get on top of 2 decades worth of research?

Breivold et al.'s Systematic Review

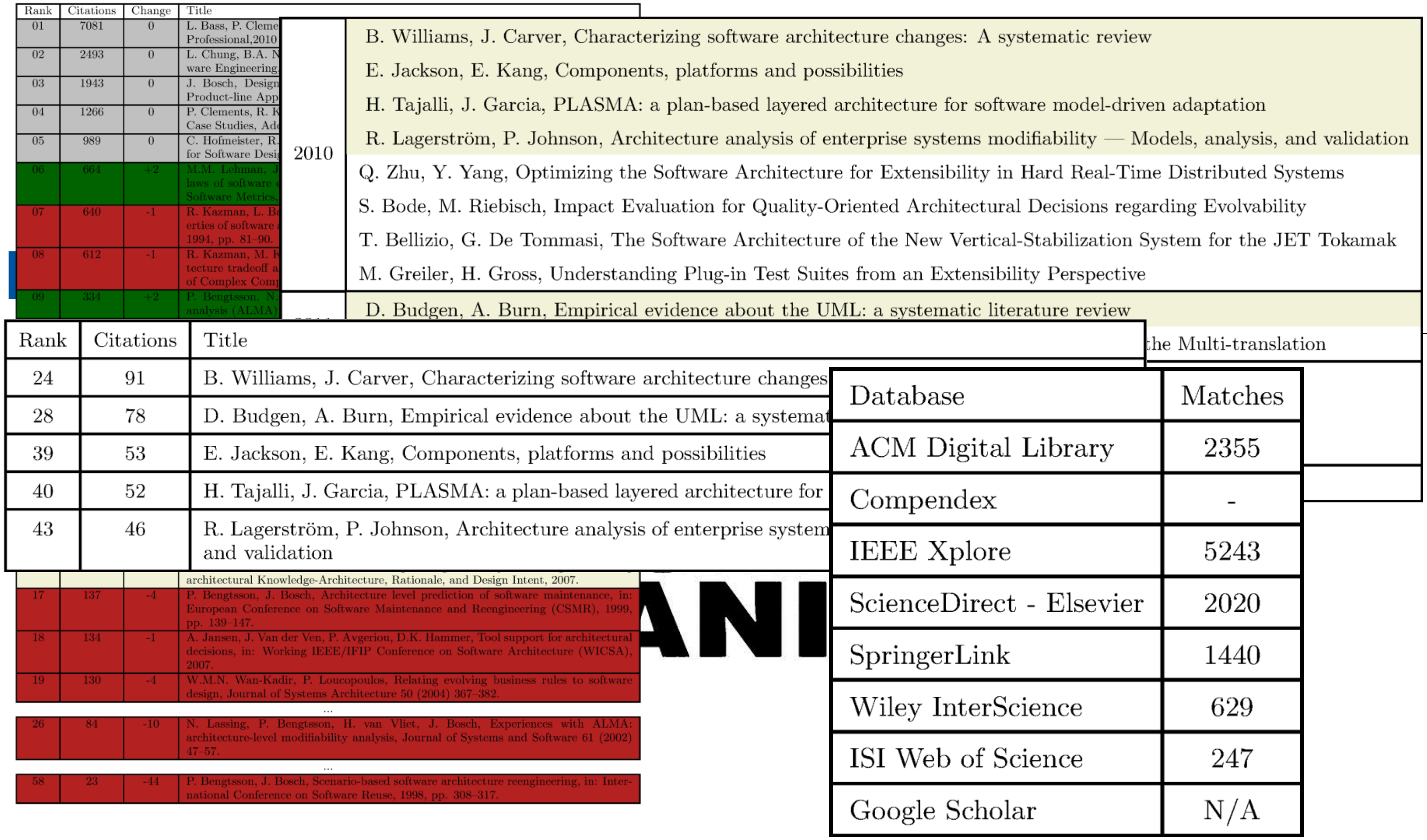


RWTHAACHEN UNIVERSITY

Breivold et al. – Systematic Review

Updating Browold for 2016

RWTHAACHEN UNIVERSITY



Rank	Citations	Change	Title
01	7081	0	L. Bass, P. Cleme Professional,2010
02	2493	0	L. Chung, B.A. N ware Engineering,
03	1943	0	J. Bosch, Design Product-line App
04	1266	0	P. Clements, R. K Case Studies, Add
05	989	0	C. Hofmeister, R. for Software Desi
06	664	+2	M.M. Lehman, J laws of software Software Metrics,
07	640	-1	R. Kazman, L. Ba erties of software 1994, pp. 81–90.
08	612	-1	R. Kazman, M. K tecture tradeoff a of Complex Comp
09	334	+2	P. Bengtsson, N. analysis (ALMA)
			architectural Knowledge-Architecture, Rationale, and Design Intent, 2007.
17	137	-4	P. Bengtsson, J. Bosch, Architecture level prediction of software maintenance, in: European Conference on Software Maintenance and Reengineering (CSMR), 1999, pp. 139–147.
18	134	-1	A. Jansen, J. Van der Ven, P. Avgeriou, D.K. Hammer, Tool support for architectural decisions, in: Working IEEE/IFIP Conference on Software Architecture (WICSA), 2007.
19	130	-4	W.M.N. Wan-Kadir, P. Loucopoulos, Relating evolving business rules to software design, Journal of Systems Architecture 50 (2004) 367–382.

...

26	84	-10	N. Lassing, P. Bengtsson, H. van Vliet, J. Bosch, Experiences with ALMA: architecture-level modifiability analysis, Journal of Systems and Software 61 (2002) 47–57.

...

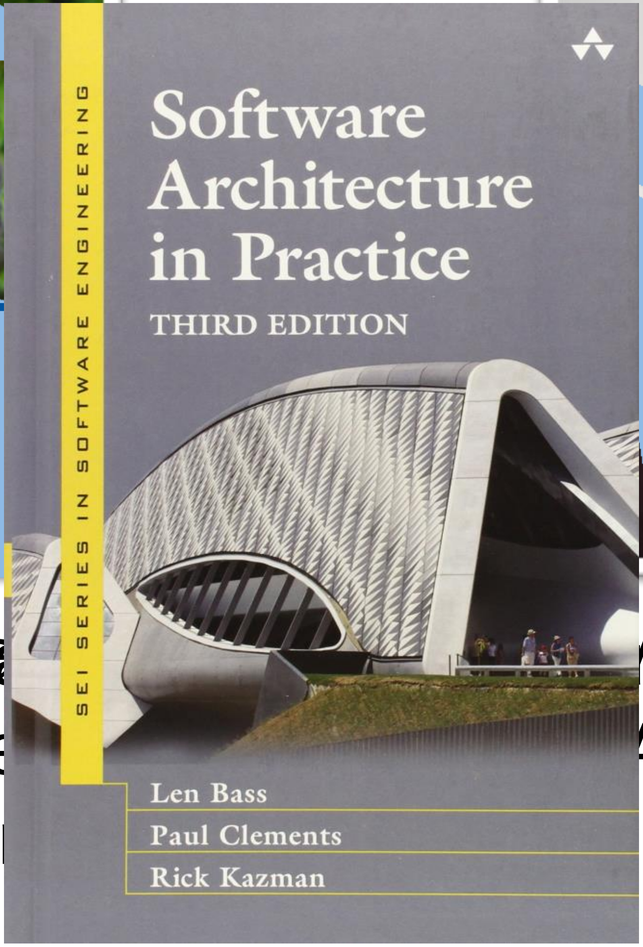
58	23	-44	P. Bengtsson, J. Bosch, Scenario-based software architecture reengineering, in: International Conference on Software Reuse, 1998, pp. 308–317.

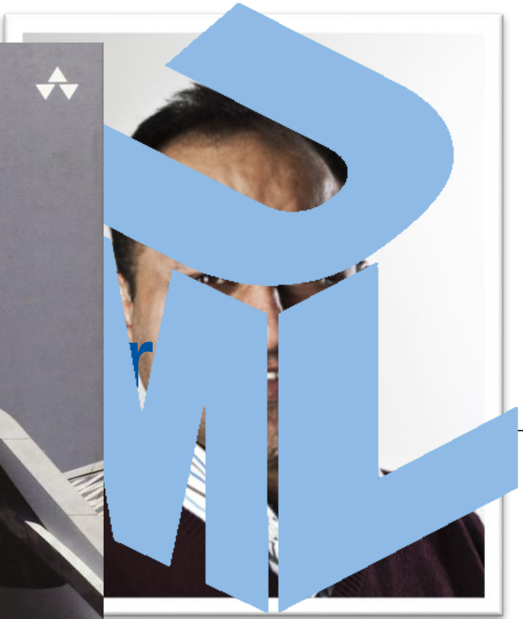
2010	B. Williams, J. Carver, Characterizing software architecture changes: A systematic review
	E. Jackson, E. Kang, Components, platforms and possibilities
	H. Tajalli, J. Garcia, PLASMA: a plan-based layered architecture for software model-driven adaptation
	R. Lagerström, P. Johnson, Architecture analysis of enterprise systems modifiability — Models, analysis, and validation
	Q. Zhu, Y. Yang, Optimizing the Software Architecture for Extensibility in Hard Real-Time Distributed Systems
	S. Bode, M. Riebisch, Impact Evaluation for Quality-Oriented Architectural Decisions regarding Evolvability
	T. Bellizio, G. De Tommasi, The Software Architecture of the New Vertical-Stabilization System for the JET Tokamak
	M. Greiler, H. Gross, Understanding Plug-in Test Suites from an Extensibility Perspective
	D. Budgen, A. Burn, Empirical evidence about the UML: a systematic literature review
	the Multi-translation

Rank	Citations	Title
24	91	B. Williams, J. Carver, Characterizing software architecture changes
28	78	D. Budgen, A. Burn, Empirical evidence about the UML: a systemat
39	53	E. Jackson, E. Kang, Components, platforms and possibilities
40	52	H. Tajalli, J. Garcia, PLASMA: a plan-based layered architecture for
43	46	R. Lagerström, P. Johnson, Architecture analysis of enterprise system and validation

Database	Matches
ACM Digital Library	2355
Compendex	-
IEEE Xplore	5243
ScienceDirect - Elsevier	2020
SpringerLink	1440
Wiley InterScience	629
ISI Web of Science	247
Google Scholar	N/A

Who and Wha


SEI SERIES IN SOFTWARE ENGINEERING
Software Architecture in Practice
THIRD EDITION
Len Bass
Paul Clements
Rick Kazman



Rick Kazma

Architectu
Knowledge
Manageme

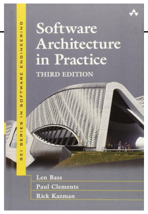
Mohammad Ali
Babar

Model-Driven
Architecture

Some Considerations

- Breivold's Review:

Conclusions and Final Thoughts

- New and old Trends: 

Software Architecture in Practice
THIRD EDITION
Len Bass
Paul Clements
Rick Kazman

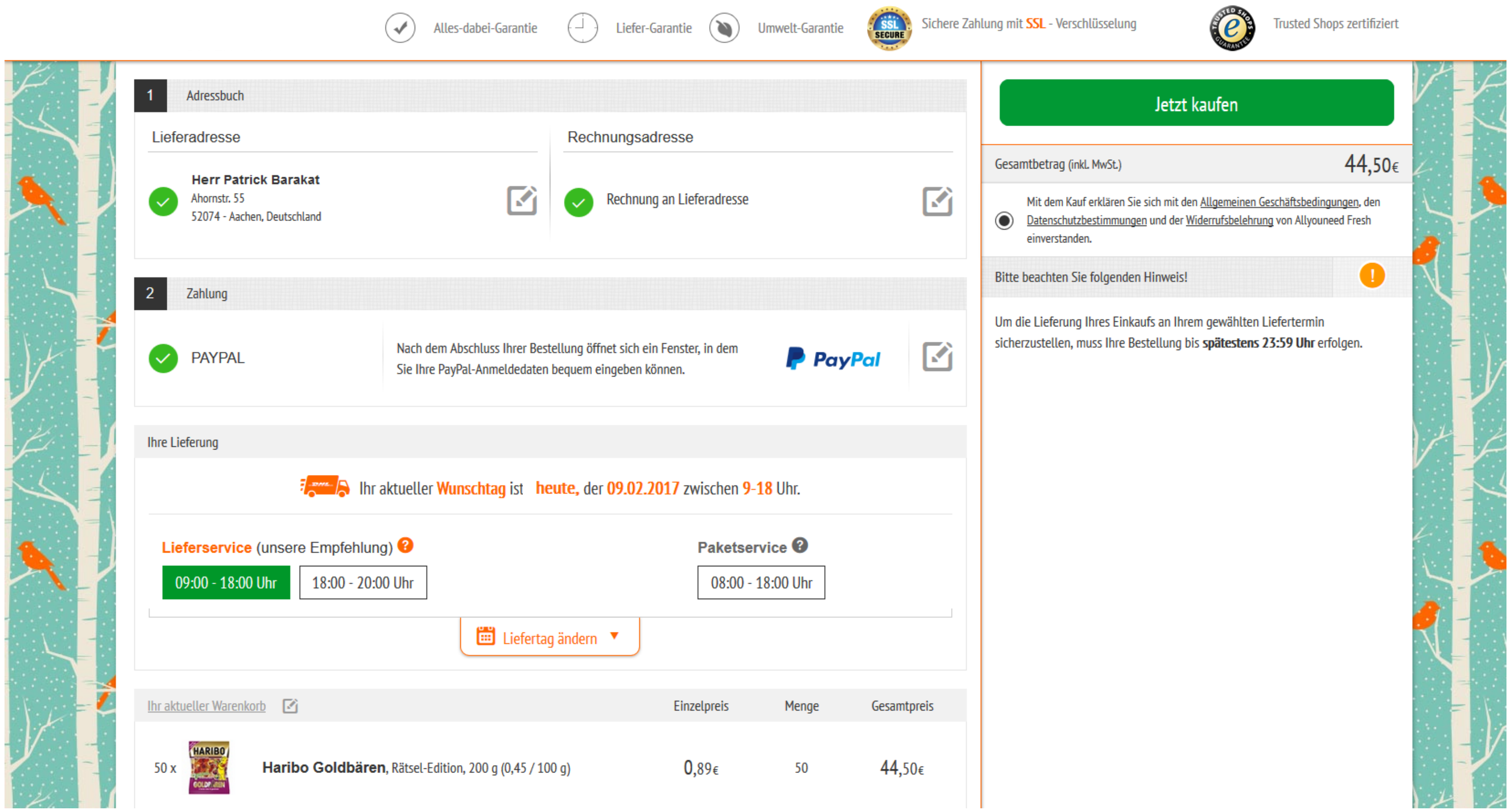
- But:

Patrick Barakat,
patrick.Barakat@rwth-aachen.de

Black-Box Testing in the Presence of Database Inputs

SWC 2017 – The Art of Software Testing

Order process example



Alles-dabei-Garantie Liefer-Garantie Umwelt-Garantie Sichere Zahlung mit SSL - Verschlüsselung Trusted Shops zertifiziert

1 Adressbuch

Lieferadresse

Herr Patrick Barakat
Ahornstr. 55
52074 - Aachen, Deutschland

Rechnungsadresse

Rechnung an Lieferadresse

2 Zahlung

PAYPAL

Nach dem Abschluss Ihrer Bestellung öffnet sich ein Fenster, in dem Sie Ihre PayPal-Anmeldedaten bequem eingeben können.

PayPal

Ihre Lieferung

Ihr aktueller Wunschtag ist heute, der 09.02.2017 zwischen 9-18 Uhr.

Lieferservice (unsere Empfehlung)

09:00 - 18:00 Uhr | 18:00 - 20:00 Uhr

Paketservice

08:00 - 18:00 Uhr

Liefertag ändern

Ihr aktueller Warenkorb		Einzelpreis	Menge	Gesamtpreis
50 x	**Haribo Goldbären**, Rätsel-Edition, 200 g (0,45 / 100 g)	0,89€	50	44,50€

Jetzt kaufen

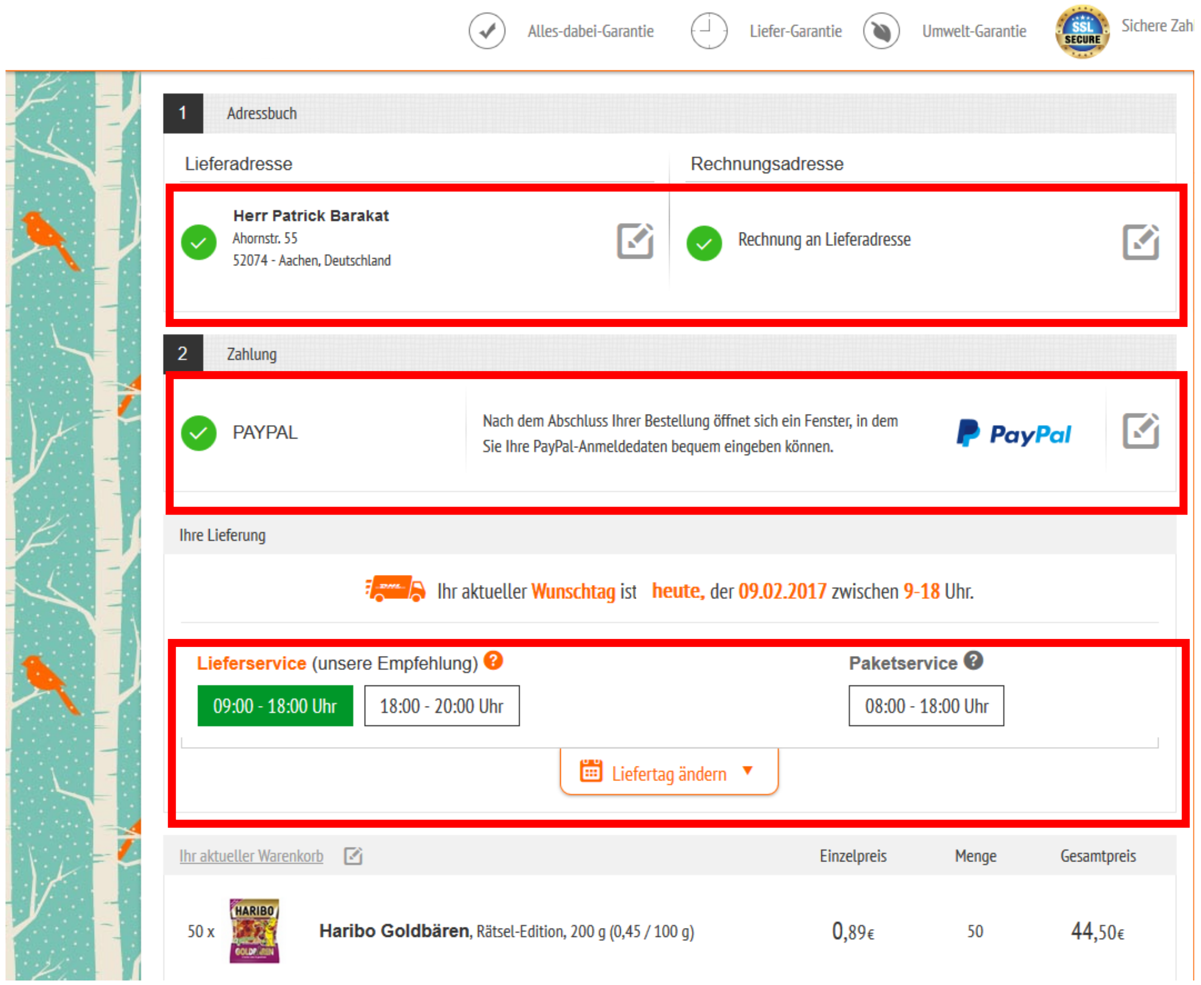
Gesamtbetrag (inkl. MwSt.) 44,50€

Mit dem Kauf erklären Sie sich mit den Allgemeinen Geschäftsbedingungen, den Datenschutzbestimmungen und der Widerrufsbelehrung von Allyouneed Fresh einverstanden.

Bitte beachten Sie folgenden Hinweis!

Um die Lieferung Ihres Einkaufs an Ihrem gewählten Liefertermin sicherzustellen, muss Ihre Bestellung bis **spätestens 23:59 Uhr** erfolgen.

Order process example – Input data


Alles-dabei-Garantie Liefer-Garantie Umwelt-Garantie SSL SECURE Sichere Zah

1 Adressbuch

Lieferadresse	Rechnungsadresse
Herr Patrick Barakat Ahornstr. 55 52074 - Aachen, Deutschland	Rechnung an Lieferadresse

2 Zahlung

PAYPAL — Nach dem Abschluss Ihrer Bestellung öffnet sich ein Fenster, in dem Sie Ihre PayPal-Anmeldedaten bequem eingeben können. PayPal

Ihre Lieferung

Ihr aktueller Wunschtag ist heute, der 09.02.2017 zwischen 9-18 Uhr.

Lieferservice (unsere Empfehlung): 09:00 - 18:00 Uhr | 18:00 - 20:00 Uhr

Paketservice: 08:00 - 18:00 Uhr

Liefertag ändern

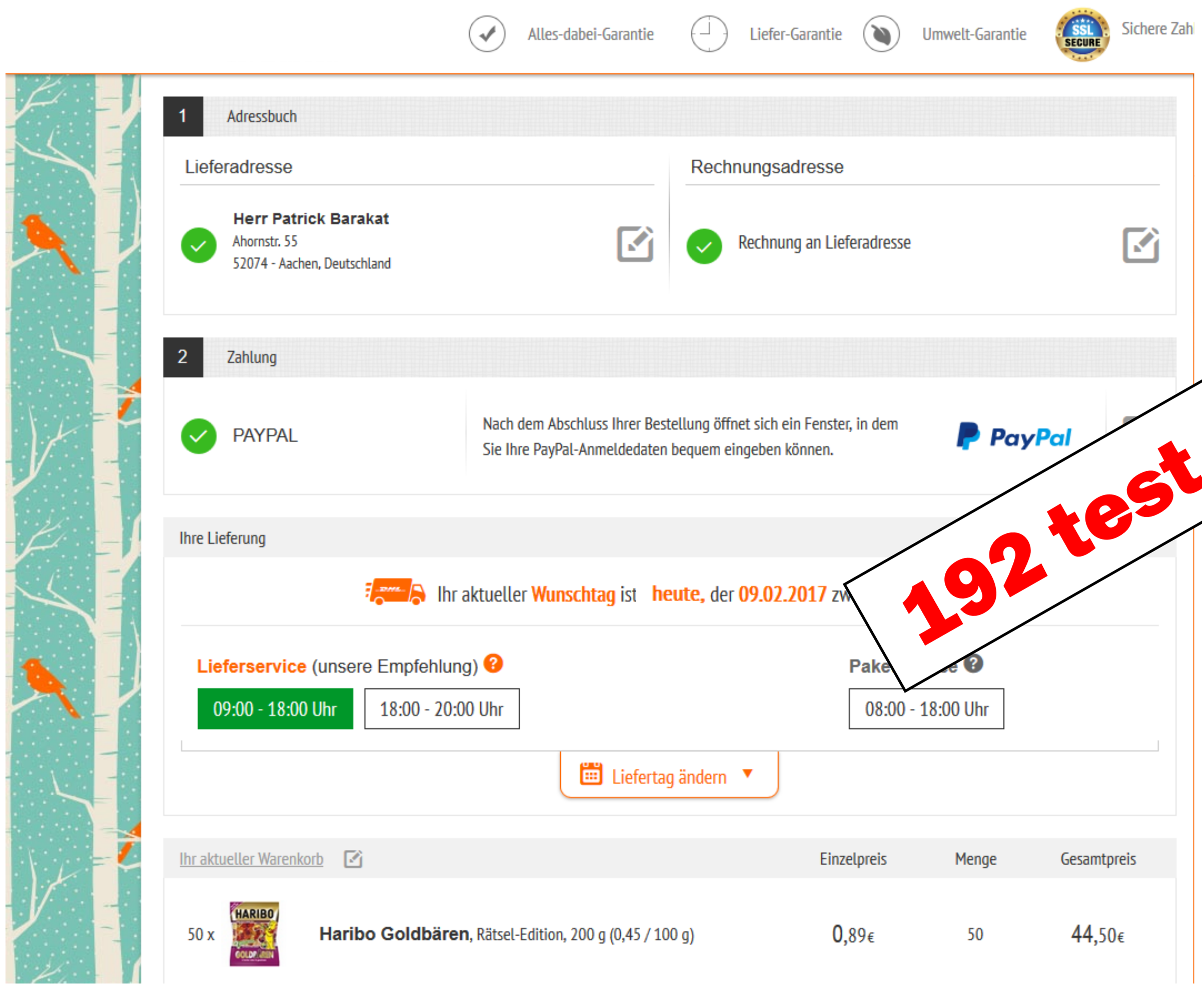
Ihr aktueller Warenkorb		Einzelpreis	Menge	Gesamtpreis
50 x	Haribo Goldbären, Rätsel-Edition, 200 g (0,45 / 100 g)	0,89€	50	44,50€

Shipping address:	German address, Foreign address
Billing address:	German address, Foreign address

Payment method:	Paypal, Debit, Payment in advance, not specified

Delivery time:	09:00 – 18:00 18:00 – 20:00 08:00 – 18:00
Delivery day:	{1}, {2}, { 3,…,30}, {30, 31,…}

Order process example – Input data


Alles-dabei-Garantie Liefer-Garantie Umwelt-Garantie Sichere Zah

1 Adressbuch

Lieferadresse

Herr Patrick Barakat
Ahornstr. 55
52074 - Aachen, Deutschland

Rechnungsadresse

Rechnung an Lieferadresse

2 Zahlung

PAYPAL

Nach dem Abschluss Ihrer Bestellung öffnet sich ein Fenster, in dem Sie Ihre PayPal-Anmeldedaten bequem eingeben können.

Ihre Lieferung

Ihr aktueller Wunschtag ist heute, der 09.02.2017 zw

Lieferservice (unsere Empfehlung)

09:00 - 18:00 Uhr 18:00 - 20:00 Uhr

Pake

08:00 - 18:00 Uhr

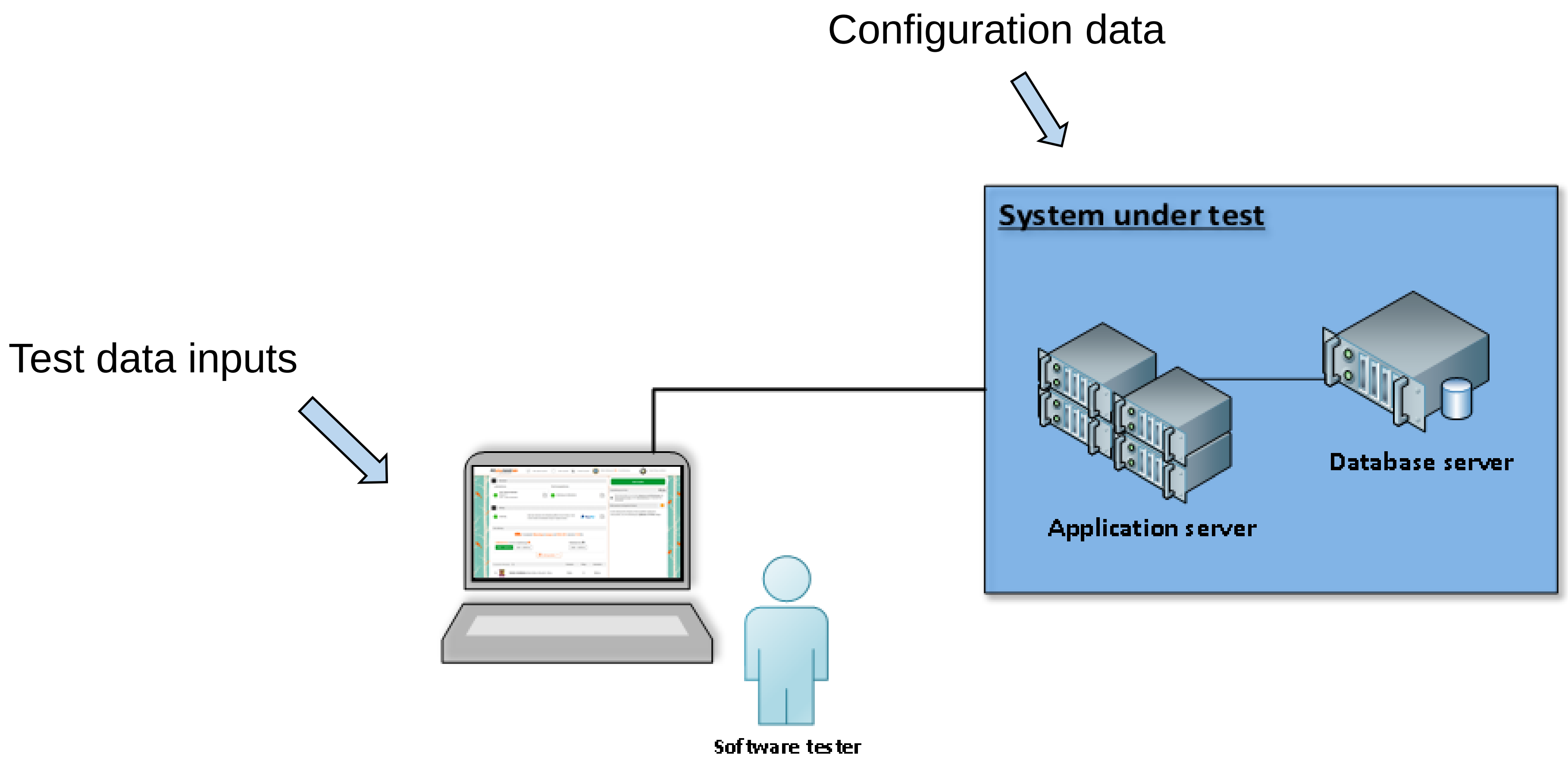
Liefertag ändern

Ihr aktueller Warenkorb		Einzelpreis	Menge	Gesamtpreis
50 x	Haribo Goldbären, Rätsel-Edition, 200 g (0,45 / 100 g)	0,89€	50	44,50€

Shipping address:	German address, Foreign address
Billing address:	German address, Foreign address
yment method:	Paypal, Debit, Payment in advance, not specified
Delivery time:	09:00 – 18:00 18:00 – 20:00 08:00 – 18:00
Delivery day:	{1}, {2}, { 3,…,30}, {30, 31,…}

192 test cases

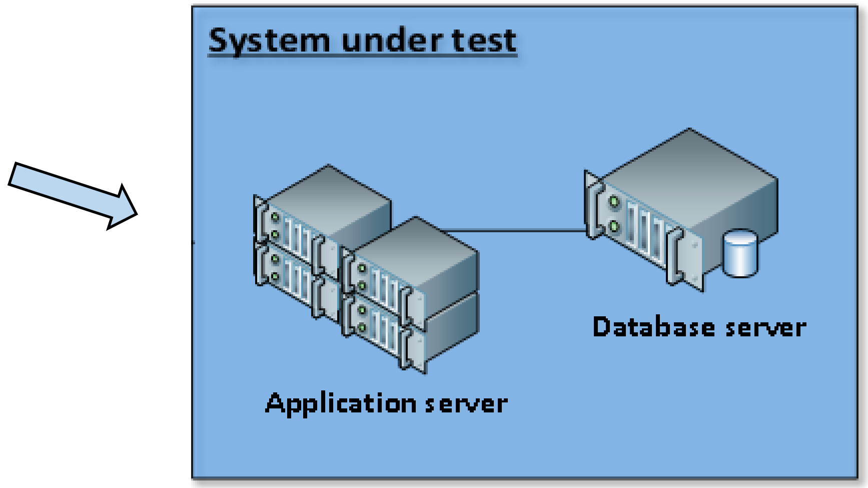
Ways of using combinatorial testing



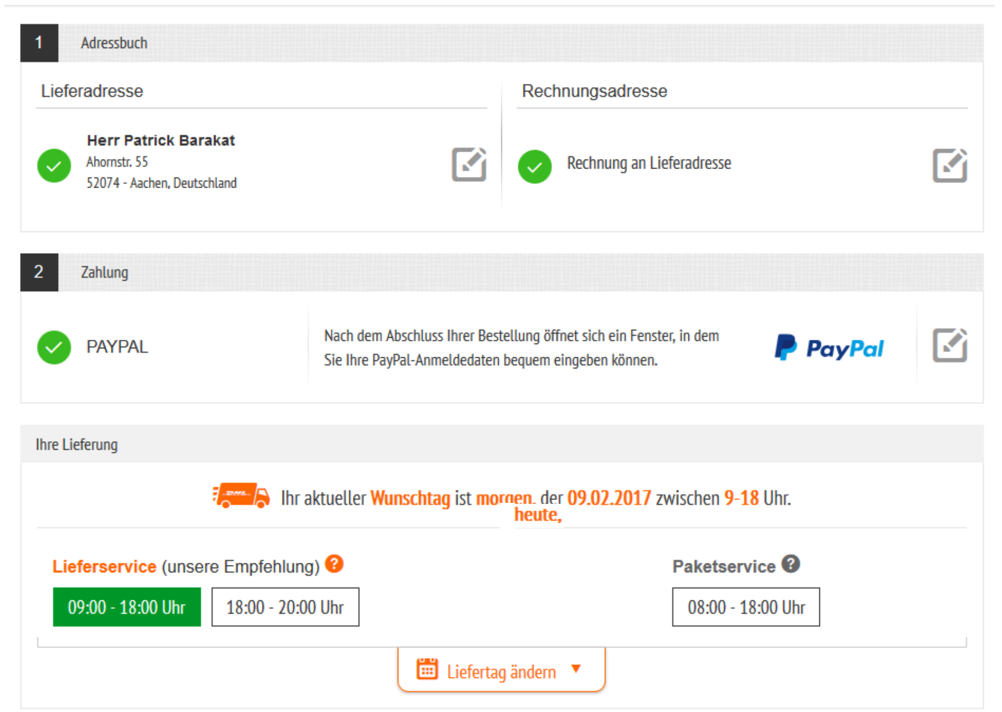
Order process example – Sample values (1)

Configuration data

Operating System Configuration	Database Configuration	Browser Configuration
Win XP	Oracle	IE 8
Win 7 32 bit	DB2	IE 9
Win 7 64 bit	MySql	FF 16.0
Win 8 32 bit	MSSQL Server	Chrome
Win 8 64 bit	Sybase	



Order process example – Sample values (2)



1 Adressbuch

Lieferadresse

Herr Patrick Barakat
Ahornstr. 55
52074 - Aachen, Deutschland

Rechnungsadresse

Rechnung an Lieferadresse

2 Zahlung

PAYPAL

Nach dem Abschluss Ihrer Bestellung öffnet sich ein Fenster, in dem Sie Ihre PayPal-Anmeldedaten bequem eingeben können.

Ihre Lieferung

Ihr aktueller Wunschtag ist ~~morgen~~ heute, der 09.02.2017 zwischen 9-18 Uhr.

Lieferservice (unsere Empfehlung) ?

09:00 - 18:00 Uhr | 18:00 - 20:00 Uhr

Paketservice ?

08:00 - 18:00 Uhr

Liefertag ändern

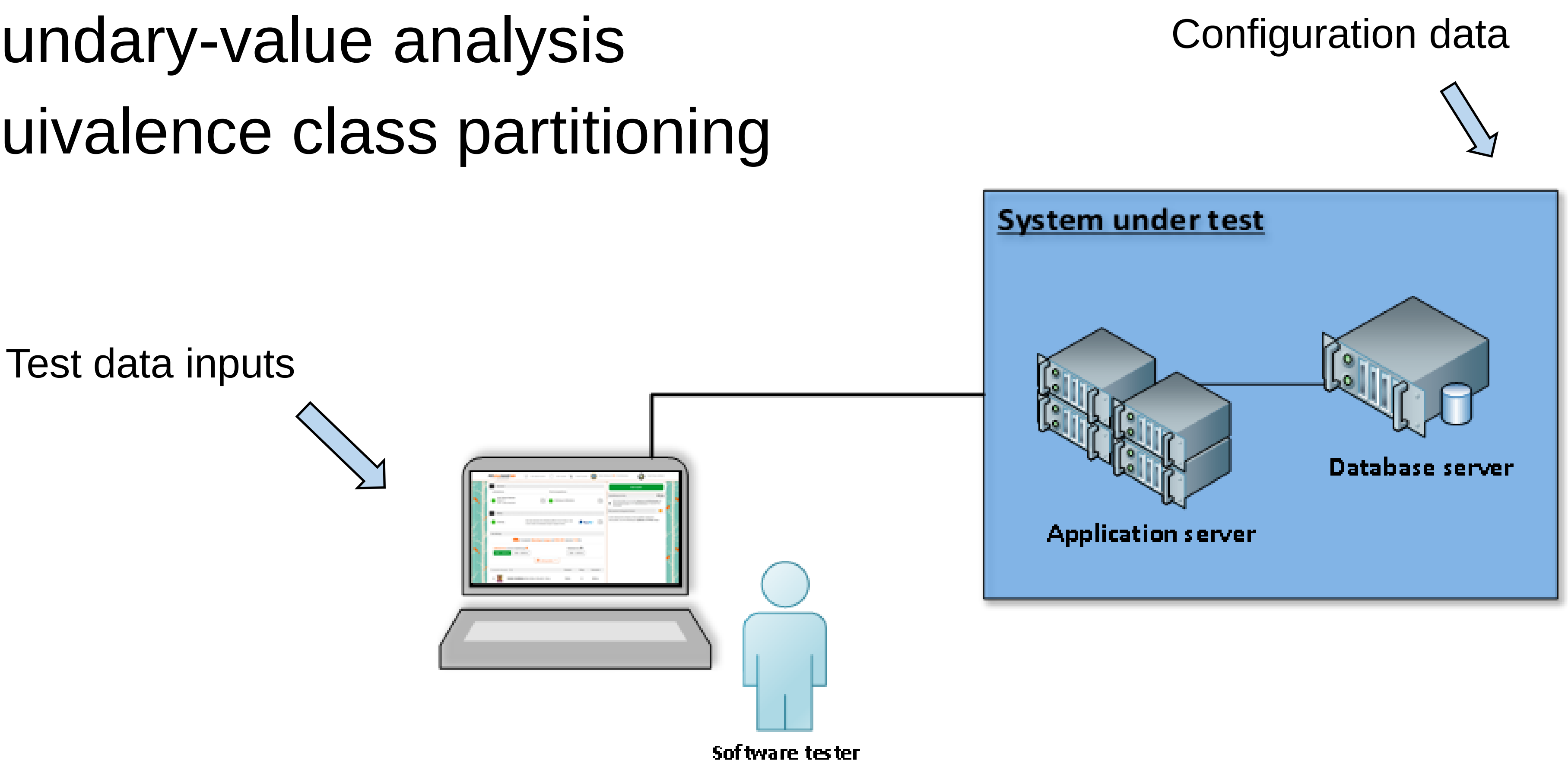
Input data

Client no.	Shipping address	Payment method
1	Angle alley 7	Paypal
2	King's Road 1	Payment in advance
3	Downing street 100	Debit
4	NULL	NULL

Definition of sample values

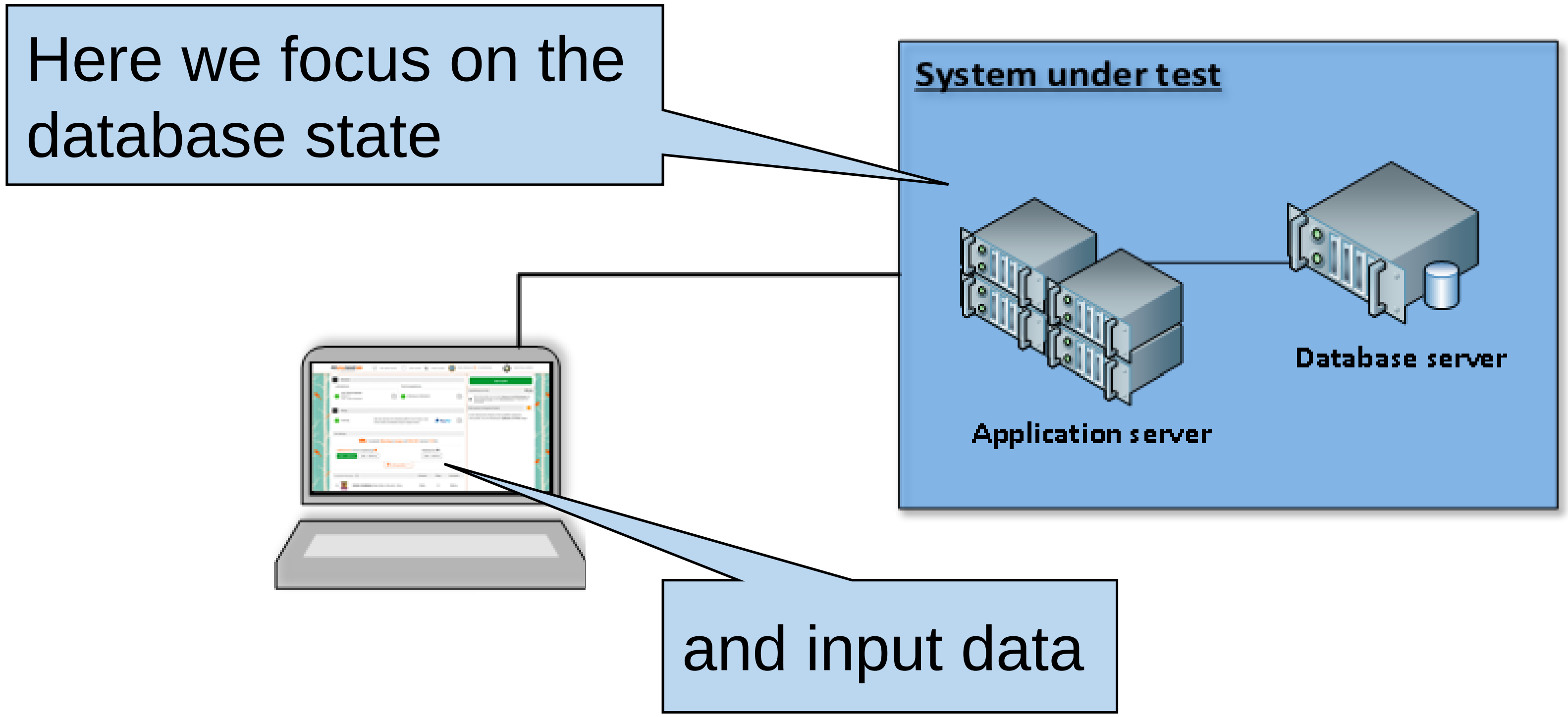
Selecting values in example by

- Boundary-value analysis
- Equivalence class partitioning



Combinatorial testing

- Generate combinations to be tested.
- State-of-the-art approaches focus usually only on one area.



Motivation

- Generate combinations to be tested.
- State-of-the-art approaches focus usually only on one area.

💡 Consider interactions between the database and the input values

⇨ Use database queries and sample values for test case reduction

Input specification

Queries

- Client that is registered
- Client that has a payment method
- Client that has *no* reminder procedure
- Client with shipping address and payment method that are *not new* and *not empty*

Input data

Client no.	Shipping address	Payment method
1	Angle alley 7	Paypal
2	King's Road 1	Payment in advance
3	Downing street 100	Debit
4	NULL	NULL

Input specification (2)

- Assume table *client* and a table *order* are already filled:

Client no.	Shipping address (address)	Payment method (payment)	Reminder procedure
1	Angle alley 7	Paypal	No
2	Downing street 100	Payment in advance	Yes
3	NULL	NULL	No

Order no.	Order value	Client id
5	500	1
10	105	2

Depedency relation

Query	Input data dependence	Allowed inputs	Disallowed inputs
Client that is registered	{client no.}	{1,2,3}	{4}
Client that has a payment method	{client no., payment}	{1,2}	{3,4}
Client that has *no* reminder procedure	{client no.}	{1,3}	{2,4}
Client with shipping address and payment method that are *not new* and *not empty*	{client no., address, payment}	{(1, Angle alley 7, Paypal), (2, Downing street 100, Payment in advance)}	{t| t=(x,y,z), x=1,..,3, t not in query 4}

Depedency relation

Query	Input data dependence	Allowed inputs	Disallowed inputs
Client that is registered	{client no.}	{1,2,3}	{4}
Client that has a payment method	{client no., payment}	{1,2}	{3,4}
Client that has *no* reminder procedure	{client no.}	{1,3}	{2,4}
Client with shipping address and payment method that are *not new* and *not empty*	{client no., address, payment}	{(1, Angle alley 7, Paypal), (2, Downing street 100, Payment in advance)}	{t| t=(x,y,z), x=1,..,3, t not in query 4}

Depedency relation

Query	Input data dependence	Allowed inputs	Disallowed inputs
Client that is registered	{client no.}	{1,2,3}	{4}
Client that has a payment method	{client no., payment}	{1,2}	{3,4}
Client that has *no* reminder procedure	{client no.}	{1,3}	{2,4}
Client with shipping address and payment method that are *not new* and *not empty*	{client no., address, payment}	{(1, Angle alley 7, Paypal), (2, Downing street 100, Payment in advance)}	{t| t=(x,y,z), x=1,..,3, t not in query 4}

Depedency relation

Query	Input data dependence	Allowed inputs	Disallowed inputs
Client that is registered	{client no.}	{1,2,3}	{4}
Client that has a payment method	{client no., payment}	{1,2}	{3,4}
Client that has *no* reminder procedure	{client no.}	{1,3}	{2,4}
Client with shipping address and payment method that are *not new* and *not empty*	{client no., address, payment}	{(1, Angle alley 7, Paypal), (2, Downing street 100, Payment in advance)}	{t| t=(x,y,z), x=1,..,3, t not in query 4}

Reduction of test cases

Query	Allowed inputs
1	{1,2,3}
2	{1,2}
3	{1,3}
4	{(1, Angle alley 7,Paypal), (2,Downing street 100, Payment in advance)}

Test	Client	Shipping address	Payment method
1.	1	Angle alley 7	Paypal

Reduction of test cases (2)

Query	Disallowed inputs
1	{4}
2	{3,4}
3	{2,4}
4	{t| t=(x,y,z),x=1,..,3, t not in query 4}

Test no.	Client no.	Shipping address	Payment method
1.	1	Angle alley 7	Paypal
2.	1	King's Road 1	Payment in advance
3.	4	King's Road 1	Debit

The idea of the approach

- Generate combinations between the sample values.
- Generate database content/queries.
- Compute dependencies between input values and queries.

⇨ All queries should be covered at least once

(Adapted) Late Acceptance Hill-Climbing algorithm

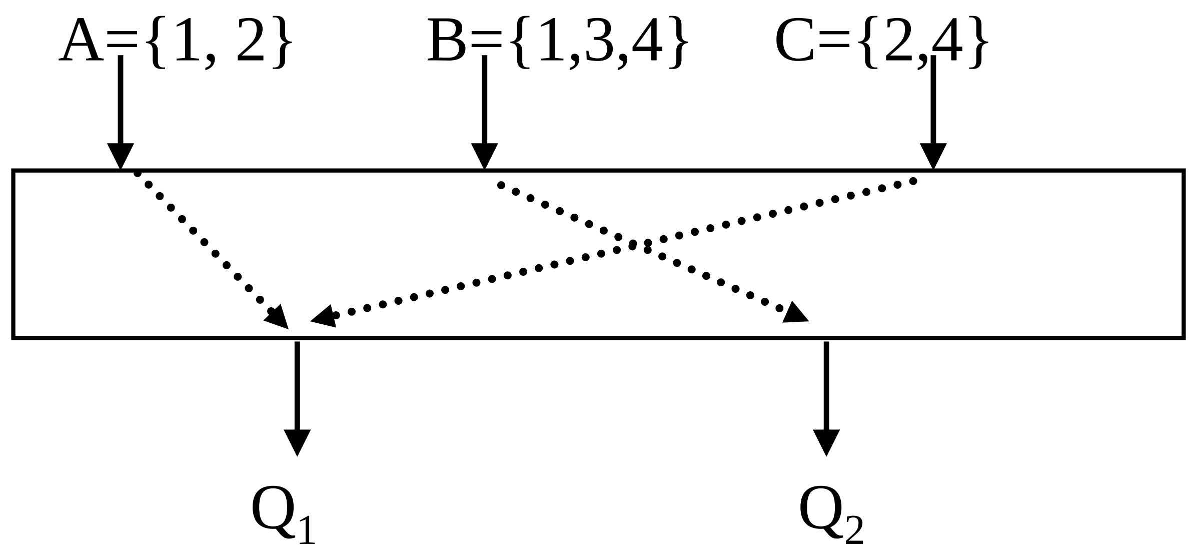
- Optimisation metaheuristic
- First LAHC-based method for combinatorial testing

- Computes a set of test cases
- Maximize the set of covered interactions

- Incorporate constraints of parameter values

Input-Output-Analysis

- Cover all combinations in the individual output test sets
- Reduce test cases by input/query relation



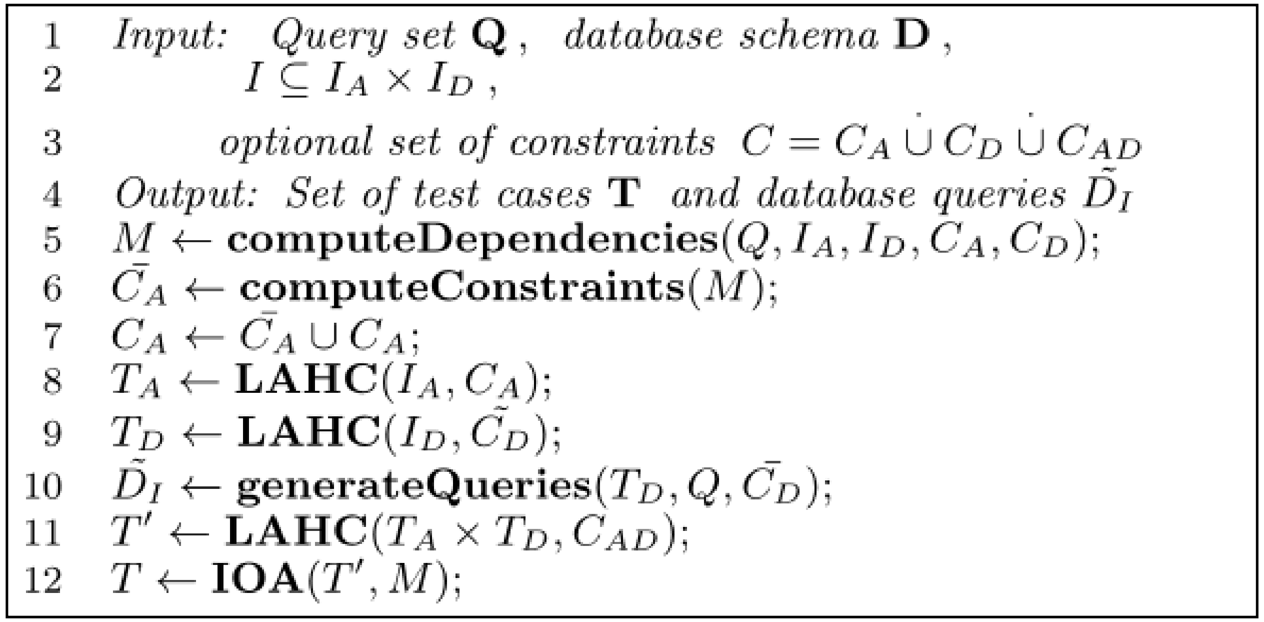
Algorithm in pseudo-code

Input:

- A set of queries,
- a database schema,
- sample values,
- and constraints.

Output:

- A set of test cases
- and queries

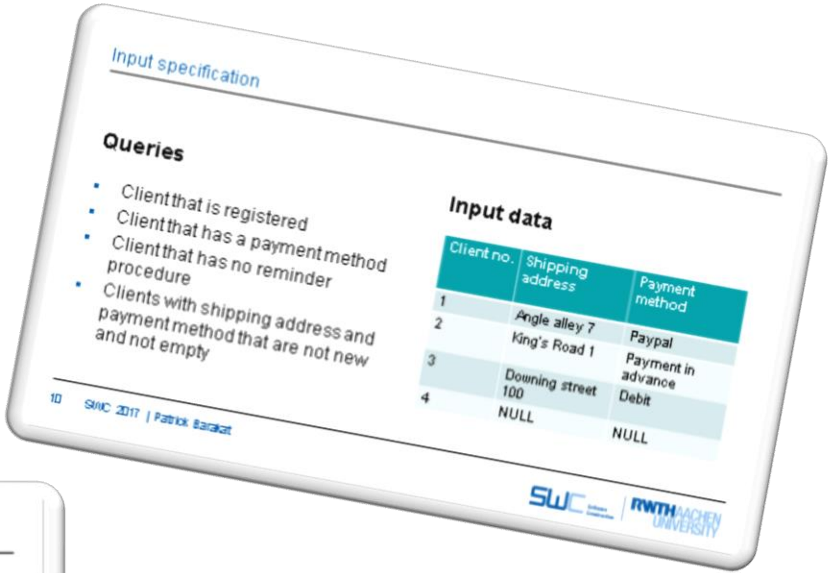


```
Input: Query set $\mathbf{Q}$, database schema $\mathbf{D}$,
        $I \subseteq I_A \times I_D$,
        optional set of constraints $C = C_A \dot\cup C_D \dot\cup C_{AD}$
Output: Set of test cases $\mathbf{T}$ and database queries $\tilde{D}_I$
$M \leftarrow \textbf{computeDependencies}(Q, I_A, I_D, C_A, C_D)$;
$\bar{C}_A \leftarrow \textbf{computeConstraints}(M)$;
$C_A \leftarrow \bar{C}_A \cup C_A$;
$T_A \leftarrow \textbf{LAHC}(I_A, C_A)$;
$T_D \leftarrow \textbf{LAHC}(I_D, \tilde{C}_D)$;
$\tilde{D}_I \leftarrow \textbf{generateQueries}(T_D, Q, \bar{C}_D)$;
$T' \leftarrow \textbf{LAHC}(T_A \times T_D, C_{AD})$;
$T \leftarrow \textbf{IOA}(T', M)$;
```

Summary

Order process example

2 SWC 2017 | Patrick Barakat



Input specification

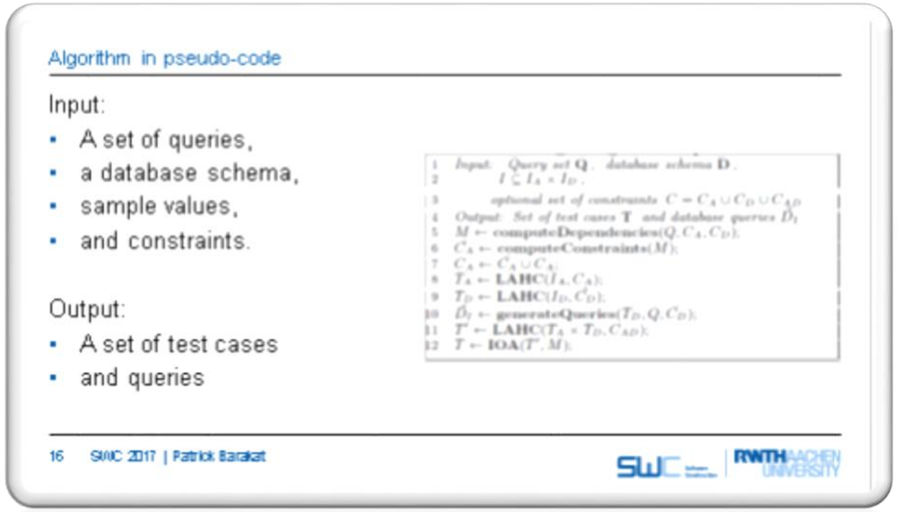
Queries

- Client that is registered
- Client that has a payment method
- Client that has no reminder procedure
- Clients with shipping address and payment method that are not new and not empty

Input data

Client no.	Shipping address	Payment method
1	Angle alley 7	Paypal
2	King's Road 1	Payment in advance
3	Downing street 100	Debit
4	NULL	NULL

10 SWC 2017 | Patrick Barakat



Algorithm in pseudo-code

Input:
- A set of queries,
- a database schema,
- sample values,
- and constraints.

Output:
- A set of test cases
- and queries

```
Input: Query set $\mathbf{Q}$, database schema $\mathbf{D}$,
        $I \subseteq I_A \times I_D$,
        optional set of constraints $C = C_A \cup C_D \cup C_{AD}$
Output: Set of test cases $\mathbf{T}$ and database queries $\hat{D}_I$
$M \leftarrow$ computeDependencies($Q, C_A, C_D$);
$C'_A \leftarrow$ computeConstraints($M$);
$C_A \leftarrow C'_A \cup C_A$;
$T_A \leftarrow$ LAHC($I_A, C_A$);
$T_D \leftarrow$ LAHC($I_D, C'_D$);
$\hat{D}_I \leftarrow$ generateQueries($T_D, Q, C_D$);
$T' \leftarrow$ LAHC($T_A \times T_D, C_{AD}$);
$T \leftarrow$ IOA($T', M$);
```

16 SWC 2017 | Patrick Barakat

Karl Ricken,

karl.ricken@rwth-aachen.de

Prioritize Test Cases the Smart Way

kinder Riegel
+MILCH -KAKAO

MilkyWay

Nestlé
KitKat
Cocoa Plan

favourite

Mars

Daim

50g
kcal 242 12%*
SNICKERS

kinder bueno
2 Riegel einzeln verpackt

If there is not enough time…

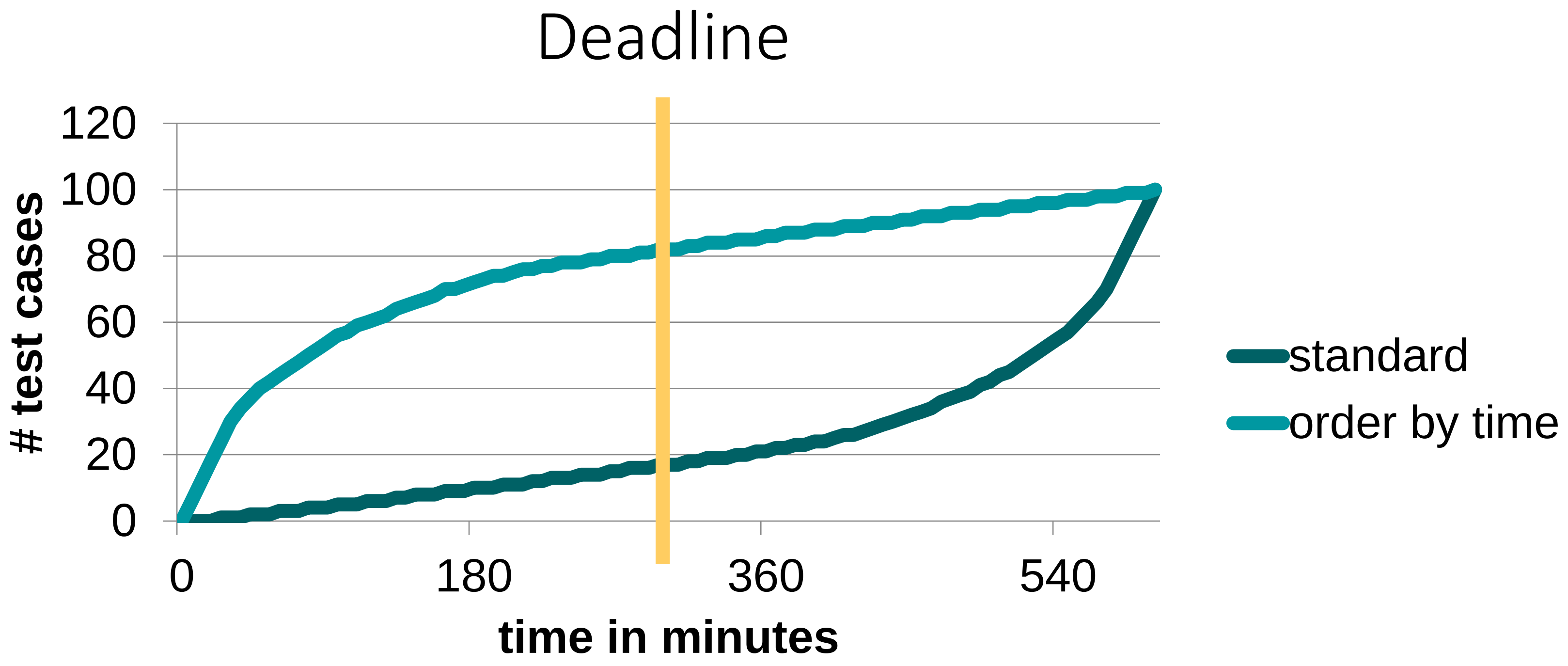
ordering: the best first

Testing Takes Time..

- Automated test to run test cases without developer
- Big software -> many test cases
- Test all test cases after code change

Prioritization

- Test as much as possible
- Get feedback as fast as possible



Order Test Cases

- Critical
- Code change relevant
- Overall time

Objectives

test critical test cases

code change relevant test cases

number of test cases

maximize

benefits

overall time

minimize

costs

How to Combine?

$$\text{maximize} \quad \frac{\text{benefit}}{\text{cost}}$$

Different Power for Test Case

- Relevance not always the same
- Weights between 0 and 1

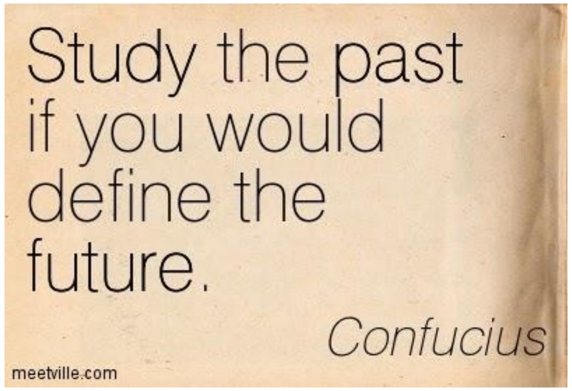
$$maximize \quad \frac{w_1 \ \ b}{w_2 \ \ c}$$

Combining More…

$$\mathit{maximize} \quad \frac{\sum w_i b_i}{\sum w_j c_j}$$

Consider More Objectives: Fault Model Sensitivity

- Execution time & critical test cases
- Learn about error in history
- Group test cases
- Prioritize new test case in group with high error rate


Study the past if you would define the future.

Confucius

meetville.com

Consider More Objectives: Setup Costs

- More objectives not related to execution
- Setup test environment

Conclusion

- State-of-the-art in the paper
- Future research for more objectives
- Combining more objectives

One more thing.

just kidding!

Good luck for the exams!

David Duong
david.duong@rwth-aachen.de

Modelling Architectural Complexity

An Overview

Architecture Complexity

Architectural Complexity

Complexity

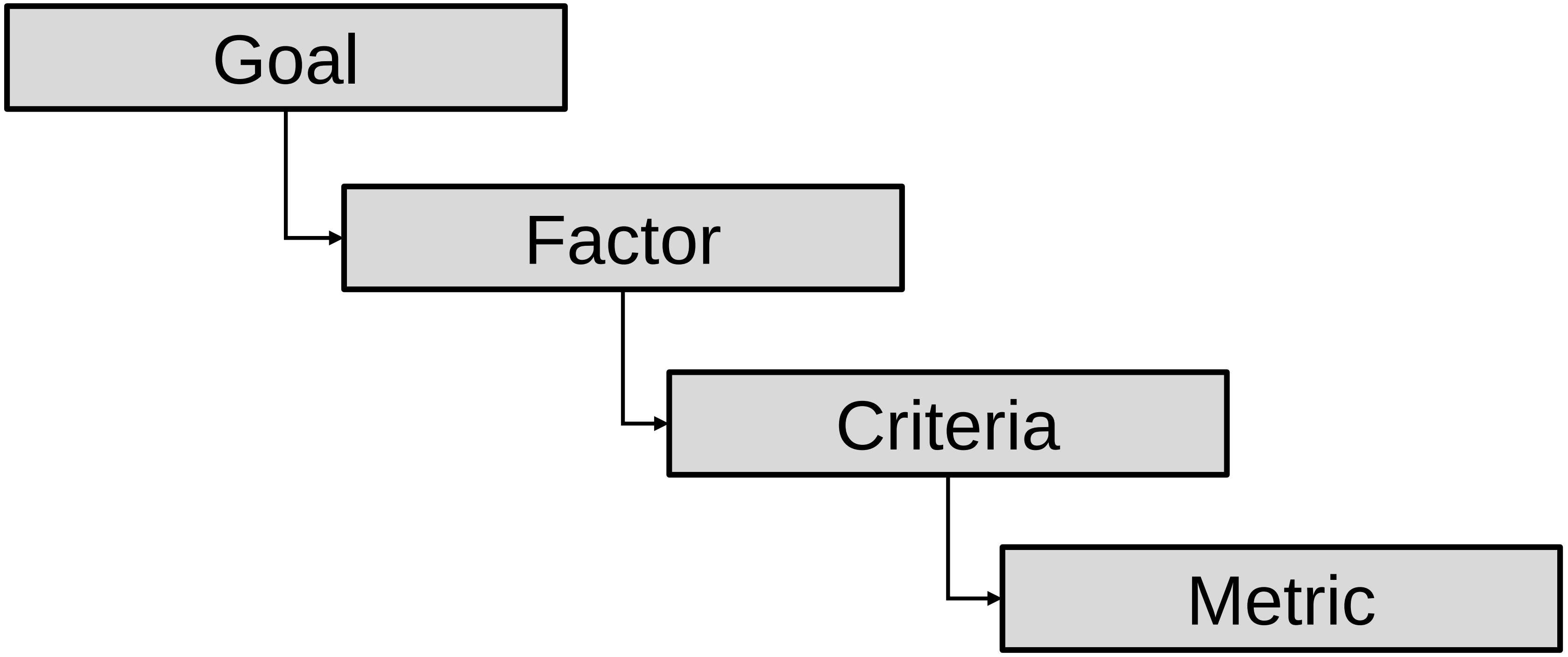
1) the degree to which a system's design or code is difficult to understand because of numerous components or relationships among components
2) pertaining to any of a set of structure-based metrics that measure the attribute in (1)

IEEE - Systems and software engineering - Vocabulary

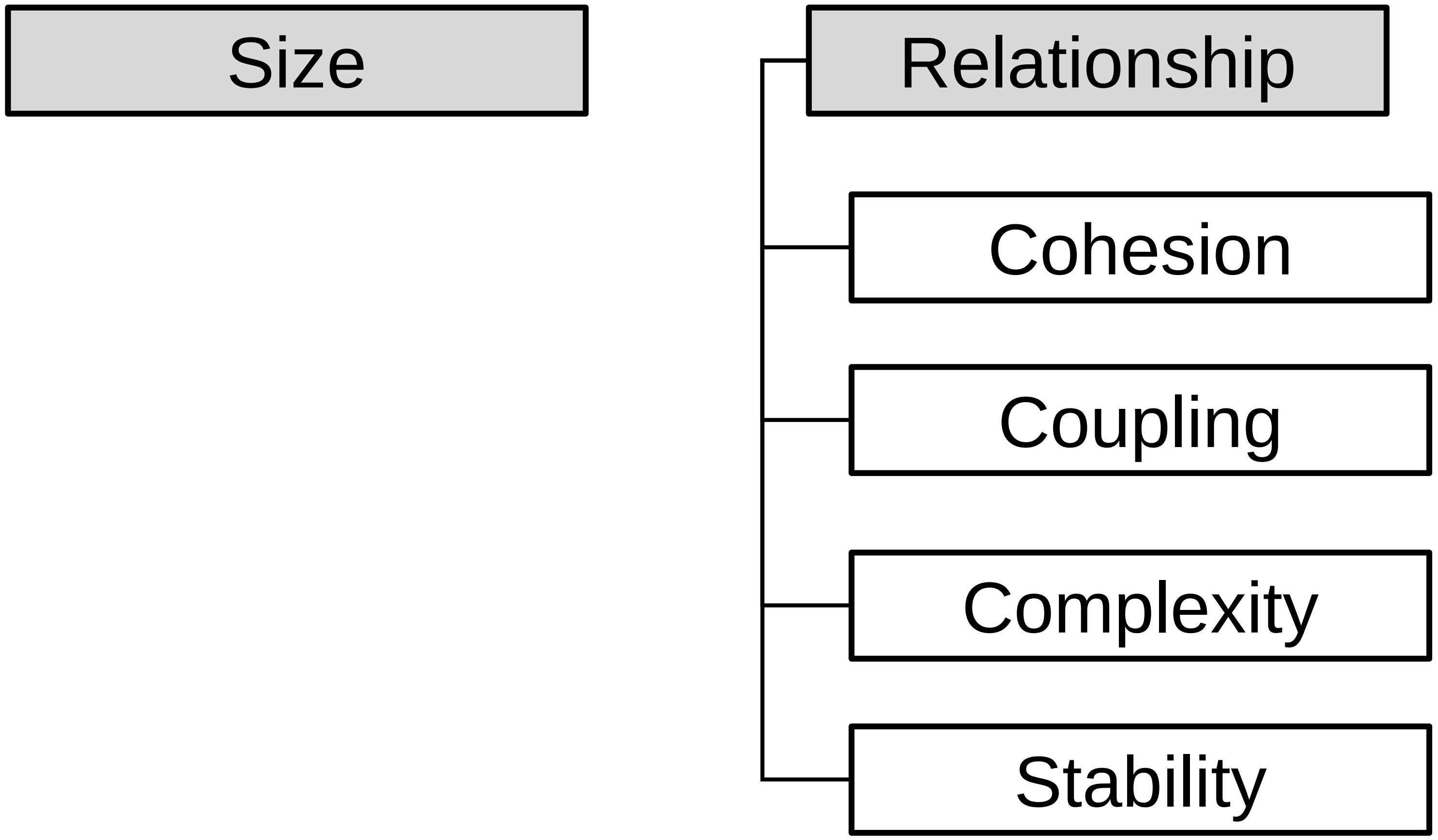
Measured Attributes

- Size
- Coupling
- Cohesion
- Complexity
- Stability
- Quality

By Srdjan Stevanetic and Uwe Zdun


Goal
Factor
Criteria
Metric

Architectural Complexity



Architecural Complexity

is the structural part of complexity of a solution, which is a result of design decisions of architectural elements and relationships among them

Lilienthal

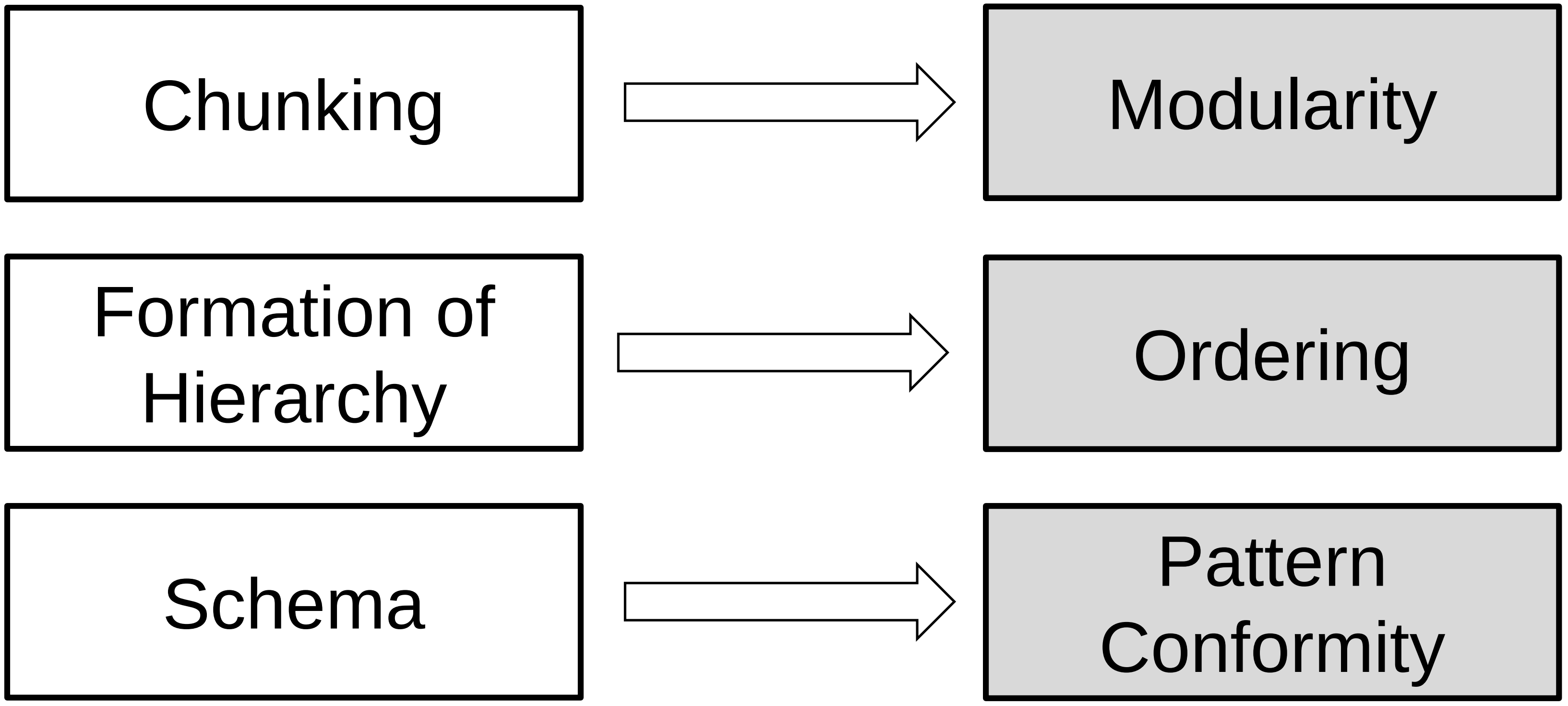
Cognitive Science

Chunking

Formation of Hierarchy

Schema

Factors



Lilienthal

Architectural Complexity

Modularity

Ordering

Pattern Conformity

Personal Factors

Information Extent

Information Availability

Enviromental Factors

Eric Bouwers et al.

Summary

- definition of architectural complexity
- modeling architecural complexity
- SACM as theoretical model based on cognitive science

- **lack of metrics**

Thank you!