Exploring interactive application landscape visualizations based on low-code automation

Nick Jansen

First supervisor: dr. ir. J.M.E.M van der Werf, UU
Second supervisor: prof. dr. S. Brinkkemper, UU
External supervisor: ir. A. Koelewijn, Mendix B.V.

CHAPTER 1

Introduction

"Software is eating the world", Marc Andreessen famously wrote in 2011[1]. His article addressed that almost every company must become a software company to stay in business. The largest traditional businesses in many domains have all been taken over or changed into software companies. Today the largest bookseller (Amazon), video service (Netflix), music company (Spotify), taxi company (Uber), hotel chain (Airbnb), recruitment service (LinkedIn), direct marketing platform (Google) are all software companies. And, industries that not have not been taken over by a dominant software company are all increasingly relying on software, today, banks, insurers, telecom and logistics companies are all slowly transforming into software companies. Software has changed from a modest service provider to an enabling driver for new business models. At the same time, modern software development methods have influenced the structure of software systems. Today more and more organizations are moving to a distributed software architecture, making already complex systems even more complex [1]. Additionally, the amount of software used in enterprises is growing each year. For a large enterprise it is not uncommon to have a landscape of thousands of applications, and this amount increases each year.

The practice of enterprise architecture is promoted to manage these increasingly important, complex and large application landscapes. Enterprise architecture is concerned with designing and realizing an enterprise's organisational structure, business processes, information systems, and infrastructure by using a set of coherent principles, methods and models [2].

The majority of enterprises struggle to produce EA documentation of adequate quality. Roth et al. determined that ~77% (n=108) of EA practitioners either have to apply huge effort to collect data or their data is of poor quality [3]. The same study shows larger organisations easily run thousands of applications that cooperate to support their daily operations. Managing such large landscapes increasingly becomes more important and complex. Studies by Kaisler et al. and Farwick et al. both reported that EA documentation is considered as time consuming, expensive and error-prone [4], [5]. Similar, Winter et al. reported

[1] Wall Street Journal, August 2011

that the increasing information volume of organizations combined with the high degree of manual work during the documentation and maintenance of EA models results in a time consuming, expensive and error-prone maintenance of EA information [6].

The research domain and practice of enterprise architecture take a holistic view on the enterprise and includes a plethora of principles, methods and models on different levels of abstraction. For this thesis there will not be engaged with the complete domain of EA but just with a part of it. Because we observed that application landscapes are increasing in importance, complexity and size and enterprise architects are struggling with their documentation. This thesis will focus on the documentation of these application landscapes. The objective of the study is to determine how application landscape documentation could be improved.

CHAPTER 2

Research Approach

In order to determine how the documentation of an application landscape could be improved several research questions have been formulated. This chapter will first present and discuss these research questions to consequently elaborate on a research method that aims on answering them.

2.1 Research Questions

The main question this thesis is aims to answer is:

***MQ**: How could the documentation of an application landscape be improved?*

To guide the research several sub questions have been formulated. Once the sub questions have been answered the main research question can also be answered.

To determine how the documentation of an application landscape can be improved, there must first be establish what is currently wrong with it. The first research question is formulated to inquire this. A literature study will be performed to answer this question.

RQ1: What are the current problems with documenting an application landscape?

Once the problems are identified, both scientific literature an industry practices will be consulted to formulate potential solutions that can address the problems.

RQ2: How could the current problems with documenting an application landscape be addressed?

Based on the answers to research question one and two, research question three, four and five have been formulated. Chapter three can be consulted for the answers to the first two research questions.

Research question one and two revealed that 1) low-code applications could be an interesting information source for the automation of application landscape documentation. And 2) interactive documentation could potentially overcome the limitations of static documentation.

Because no architectural language exists that is based on these principles. First the requirements for such a language will have to be gathered, this led to the following research questions.

RQ3: What are the requirements of an interactive language to support a model-based application landscape?

To answer RQ3, we first investigate what type of views and on demand information is relevant for a practitioner when a language is designed based on low-code automation and interactivity.

RQ3.1: What type of views and on demand information should be used to interactively visualize and analyse an application landscape model for enterprise applications?

Once the type of visualizations is defined, there should be determined which application landscape elements should be included per visualization. A landscape element can be any concept relevant in an application landscape, examples are: application, API, load balancer, ESB, etc.

RQ3.2: What landscape elements should be displayed in a relevant interactive visualization?

Based on the determined on-demand information types, per element there has to be established what information is relevant for practitioners.

RQ3.3: What information should be displayed on demand for each element in a relevant interactive view?

To make sure the identified requirements are focussed on user demands, user groups for who the systems is relevant should be defined. Based on these user groups corresponding user stories should be formulated so all functionality can be traced back to user demand. This way once the system is built, we can perform a light-weight evaluation by checking if all user stories are covered in the specification.

RQ3.4: What user groups and user stories should be addressed?

When the requirements for an architectural language are specified, there is evaluated how a language based on these requirements can be constructed. To do this a specification for the language is defined, this specification should answer the following research question.

RQ4: How could an interactive language to support model-based application landscapes be constructed?

2.2 Research Methods

Each research question has been answered along its own method. Table 1 gives an overview which method was used to answer each research question.

	Literature study	ADSRM	Case study
RQ1	X		
RQ2	X		
RQ3.1		X	
RQ3.2		X	
RQ3.3		X	
RQ3.4		X	
RQ4			X

Table 1 Overview Research Methods.

2.2.1 Literature study

To answers research questions one and two a literature study has been performed. The literature study targets both scientific as industry literature. There has been chosen to include industry literature to retrieve a complete picture of the problem domain.

2.2.2 Agile Design Science Research Model

An analysis of the research questions lead to the realization that for question 3 and 5 have a *"Wicked"* nature [7], [8]. Rittel who first coined the term in the mid- 1960s defined a wicked problem as:

> *"A wicked problem is one for which each attempt to create a solution changes the understanding of the problem. Wicked problems cannot be solved in a traditional linear fashion, because the problem definition evolves as new possible solutions are considered and/or implemented."*

RQ 3 and 5 have a wicked nature because they investigate a social context, have no stopping rule, and have no right or wrong answer. Answering these questions involves weighing several interacting, sometimes conflicting interests to come to a conclusion.

To cope with the complexity of a wicked problem the Agile Design Science Research Model as proposed by K. Conboy et al. was selected [9]. This research method has incorporated agile methods in the established Design Science Research Methodology by Peffers et al. [10]. By doing so the method aims to increase the proportion of inventive IT artefacts developed. Unlike the traditional DSRM, the ADSRM supports an evolving problem definition which is in the nature of a wicked problem. Figure 1 presents an overview of the Agile Design

Science Research Model (ADSRM), everything coloured red is an extension to the traditional DSRM.

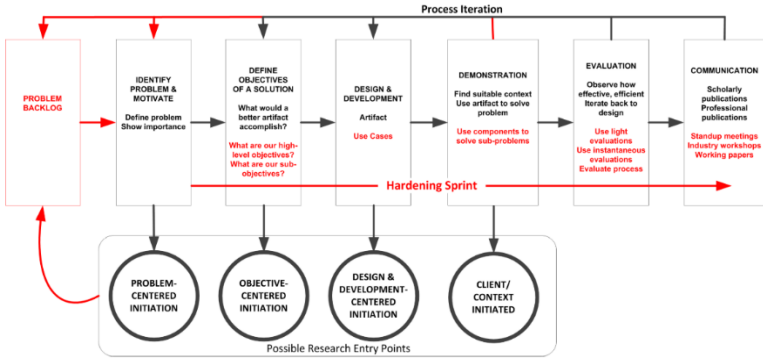


Figure 1 Agile Design Science Research Model (ADSRM).

The original DSRM consists of a project initiation and six subsequent activities. According to the DSRM a project can have four possible entry points.

- A problem-centered initiation in which little is known about a design problem.
- An objective-centered initiation where little is known how the objective of a solution impacts a problem.
- A design/development centered initiation where it is still unclear how to design a design feature.
- A client/context initiation where an industry partner invites for collaboration.

Once the project in initiated the DSRM describes six activities about how to proceed.

1. *Problem identification and motivation*. During this first step little is yet know about the problem, and first efforts are made in identifying and defining it. The problem is also motivated to ensure industrial/scientific relevance.

2. *Define the objectives for a solution*. During this activity requirements for a solution are gathered.

3. *Design and develop*. This activity involves building the actual artefact based on the earlier defined problem and objectives for the solution.

4. *Demonstration*. This activity involves testing the solution, this can be done by experimentation simulation, a case study several other means.

5. *Evaluation*. Involves determining if the proposed solution solves the defined problem.

6. Communication. Involves sharing the importance of the problem and the discovered solution. This can be done through a journal publication a presentation at a conference and many other forms.

The ADSRM adds agile concepts to each of these activities.

- To the *Define the objectives for a solution* activity there is added to think both about high and low-level objectives.
- To the *Design and develop activity* there is added that the researcher should explicitly consider the non-functional requirements.
- To the *Demonstration* activity the prescription is added that early and frequent implementations should be considered for all design concepts, not just for finished artefacts.
- To the *Evaluation* activity:
    - First light evaluations methods and metrics like *lean startup* are added.
    - Second, evaluation by instantaneous and automated testing at component level are added.
    - And third, evaluating the agility afforded by the artefact is added.
- To the *Communication* activity is added that findings should be communicated frequently inside and outside the research team.

According to the original DSRM it is only possible to iterate once the *evaluation* or *communication* activity is reached. Also, it is only possible to go back to the *define objectives of a solution* and *design & development* activity. It is not supported to go back to the *Identify problem & motivate* activity. But when dealing with a wicked problem the problem can change once an initial solution is reached. Therefore, the ADSRM grants more flexibility by allowing process iterations all the way back to the *identify problem & motivate* activity and already start iterating once the *Demonstration* activity is reached.

ADSRM also adds the concepts of a *Problem Backlog* and *Hardening Sprint* to the model. The *Problem Backlog* concept recognizes the changing problem space of design problems. By introducing a flexible problem backlog where all problems are captured more flexibility is gained while designing a solution. The *Hardening Sprint* acts as a mechanism to ensure rigour is added during the research. It does so to prevent that the agility-based amendments of the ADSRM detract from the research's rigour. Every few iterations the *Hardening Sprint* is added in the iteration process, during this sprint the focus is on enhancing rigour that might have been missing during the regular sprints. Several key mechanisms are used to accomplish this. The first one is *Freeze the Problem*, with this mechanism the problem remains fixed for a complete sprint, by not allowing turbulence, dynamism or improvisation a level of rigour can be applied. The second mechanism is *Freeze the Process*, when this mechanism is activated, during the sprint the 'people over process' principle from the agile manifesto is neglected and there will be just focused on the process. During that specific sprint there will be extra focus on adherence to procedure and compliance, improvisation will not be allowed. And the last mechanism is *Add to the Process*, during this mechanism additional rigour-driven parts can be added to the process (for example extra measures during an evaluation phase).

Because of the agile alterations we believe the ADSRM is a good fit to solve the research questions with a *"Wicket" nature.*

2.2.3 Case study

Once research question 3 is answered, sufficient knowledge is retrieved to formulate a design for an interactive language. To formulate this design an exploratory case study will be performed at a low-code vendor. This case study aims to deliver a specification for the construction of an interactive language to support model-based application landscapes.

CHAPTER 3

Literature Study

3.1 Context

One of the main motivations for this study is the increasing complexity of software systems. To thoroughly understand why software is increasing in complexity will first look at the history of software production.

In 1968, a seminal article by Melvin E. Conway was published titled “How Do Committees Invent” [11]. In the article Conway explained the close relationship between the structure of a design organization and the structure of the system it designs. He argued that organizations that produce systems are constrained to produce designs that are copies of their communication structure. This idea has far reaching implications for the management of system design and eventually would be called Conway’s law.

When this relationship would not be considered, Conway observed a certain pattern how complex system could disintegrate during development. First, the initial designer would realize the system will become large, this realization together with other pressures in the organization this will make the temptation irresistible to assign to more people to the design effort. Conway argues that it is a natural temptation for the initial designer to delegate tasks when a project is reaching his limit of comprehension. Even more so when a budget and schedule come in to play, simply because he knows he will be charged with mismanagement when he does not meet his schedule without having applied all his resources. In a design effort where the resource is human effort, this means the initial designer will be strongly incentivised to bring more people on the project. The fallacy lies in the fact that with a design effort the relation between input and output is not linear[2]. Where in for example a sewing factory placing more people behind sewing machines will linearly result in a higher output of produced clothing, this is not the case for a design effort. In addition, Conway’s law states that the size of the design organization will influence the design, resulting in a different system design. And Conway argues that this design will

[2] 7 years after Conway published his article in 1975 F. Brooks would publish his landmark book entitled “The Mythical Man-Month” elaborating on this idea about time management in software projects[43].

not be superior, from experience he knows that two men, if well-chosen will come up with a better system design than a large group.

Furthermore, already in 1968 Conway pointed out the importance of flexibility in an organization. He argued the first design is never the best possible, and therefore the system inevitably must change leading to organizational change as well. He pointed out that ways must be found to reward managers that keep their organization lean and flexible.

In 2007, McCormack, Rusnak and Baldwin took Conway's ideas to the test in an empirical study [12]. They compared software products that fulfilled the same function but were developed in two different organizational structures. At the one end was a software product developed by a commercial firm were developers were tightly coupled with respect to location, goals, structure and behaviour. And at the other end was a software product developed by an open source community were the developers were much looser coupled regarding all those aspects. They found that in all the pairs they examined the open source products were significantly more modular than the products of the commercial firms. These findings indicate a strong relationship between organizational structure and software structure.

But when Conway published his ideas in 1968 the world was not yet ready to embrace them. At the time a mechanistic management approach inspired by the ideas of Frederick Taylor was still the most influential way to structure your business [13]. His management ideas focused on standardization and efficiency. Processes and tools were considered far more important than people. Doing the work and thinking about how a certain task should be done was strictly separated. Innovation if happening at all happened only in specialized R&D departments. A mechanistic approach has worked perfectly for a production or service company with limited offerings in a stable predictable business environment. Under influence of this approach from 1945-1971 several traditional industries experienced an unprecedented rise in business activity.

By the time the question arose how to structure a software company, Taylors management ideas had proven themselves in practice for several decades. While Conway's ideas maybe have been a better fit for a software company, they just were the unproven ideas of one scientist. Therefore, Taylors mechanistic approach was also widely applied to software companies. This resulted in bureaucratic organized software companies with strict procedures, specialized departments and highly centralized authority. While a mechanistic approach is a good fit when you want to optimize for efficiency, it is not a very good fit when you must cope with rapid change. And software companies being design organizations, change was inherent to their practice.

In the nineties, there was slowly realized that this mechanistic organizational structure is a bad fit for a software company. As counterintuitive as it may sound, people realized that in a business environment in which changing requirements were certain it would be more efficient to favour agility over efficiency.

This realization lead to a new way of working now referred to as the agile movement. In 2001 the Manifesto for Agile Software Development was published pointing out four key values that should be followed in order to become more flexible as a software organization. 1) Individuals and interactions over processes and tools. 2) Working software over comprehensive documentation. 3) Customer collaboration over contract negotiation. And 4) responding to change over following a plan.

This simple manifesto had a huge impact on the way software would be produced. From 2001 on a plethora of methods which fall under the agile philosophy have been implemented at dozens of companies.

Looking back from 2018, at least two of the original signatories of the manifesto, Martin Fowler and Ron Jeffries, argue this has happened with mixed success. Fowler argues that much of what we see today is *Faux-Agile*, the result of and *Agile Industrial Complex* which is imposing processes upon teams which contradicts the agile beliefs [14]. Jeffries addresses a similar problem he calls *Dark Scrum*, he argues that when Scrum is used in practice and only the activities are implemented but the rationale of the ideas is left behind Scrum turns into Dark Scrum [15]. For agile practices to work a fundamental shift is needed in how we work and how power is distributed. Product software companies do a fairly good job in this transition. But in-house software groups at banks, governments, supply chain companies etc. have a much harder time. These organizations are usually still structured in a more or less mechanical way. For their complete enterprise to work it is undesirable to have a different management style in one department. And in those organizations (unlike in dedicated software companies) the in-house software group is just one department. Therefore, it becomes very difficult to structure it in a different manner while still being able to cooperate and comply with the rest of the organization. These enterprises might implement some agile activities, but they frequently do not fundamentally change their ways of working, resulting in *Faux-Agile* or *Dark Scrum*.

The agile movement has addressed the alignment between the business and developers and provided methods and principles to streamline their communication. This has been the first step in tearing down the mechanistic software organization.

The second step came with the introduction of DevOps. Around 2010 there was realized that for a streamlined software development life cycle (SDLC) it was not enough to only align the business with development. It does not matter how good they work together, as long as the operations department is not aligned nothing will be delivered to the user. Therefore, efforts have been made to integrate the development and the operations department. Bas, Weber and Zhu defined DevOps as *"a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality"* [16].

In a software organization, the development and operations department traditionally would be split up in two different silos. This structure has led to

some problems over the years. For example, when the developers introduce a bug that could crash the software during runtime the operations team would be responsible to get the system back online even when it is in the middle of the night. Although the developers might receive some angry messages from the operations team, they would not feel the real pain their bugs are causing. Furthermore, the operations team generally does not have the knowledge of the source code to actually solve the root cause for the bug. Often the best they can do is turn the system off and back on and ask the developer to solve their bugs. By bringing development and operations together in a team the pains and consequences of each other's work become more apparent, encouraging closer collaboration.

The challenge the DevOps movement takes on is how to integrate development and operations in a team without losing the agile characteristics it achieved the decade before. DevOps does so by focussing on automation and autonomy. Activities such as testing, configuration and deployment are automated as much as possible. The automation frees people to focus on other valuable tasks and additionally it reduces the change of human error.

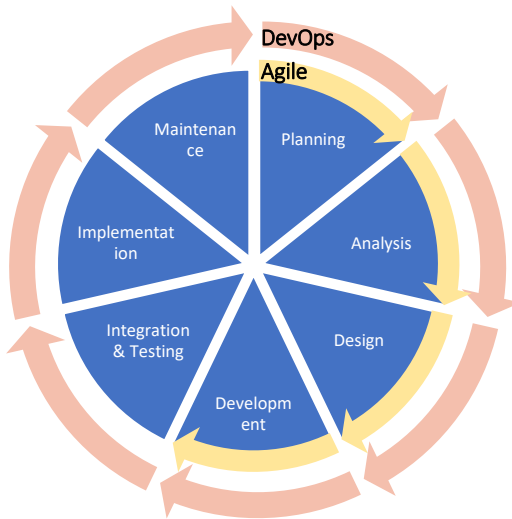


Figure 2 The Software Development Life Cycle (SDLC).

Complete team autonomy was out of reach for an agile team to achieve because everything still had to get integrated and deployed by the operations team. But with the inclusion of the operation engineers in the team it becomes possible to cover the complete software development life cycle with a single team and become autonomous. When looking at the SDLC agile focused on improving the first part of the cycle, DevOps is focussing on the complete cyle, figure 2. Although an autonomous team can be more efficient, they cannot cover a

complete system. Therefore, the system needs to be split up in smaller pieces, this way each team can be responsible for a piece. Microservices have become the de facto architectural style to split up a large system in order to achieve team autonomy [17]. With a microservice architecture the system is built as a distributed set of independently deployable services. This way each team can be given the complete responsibility over a relatively small vertical slice of the system.

But how to determine how to split up the system has been a point of elaborate discussions in the community. The Bounded Context pattern from Domain-Driven Design can help structuring the problem domain in smaller sections which can help determining microservice granularity [18]. Here we see Conway's ideas finally adopted, because the teams will be structured around a business capability and the architecture will reflect this structure, the organizational structure and system structure will finally be aligned.

Although the adoption of a microservice architecture helps teams achieving autonomy it goes at the cost of complexity [1]. Developing a distributed system is inherently more complex that developing a monolithic application. It presents challenges for availability, reliability, maintainability, performance, security and testability [1]. To address these challenges, over the last decade a lot of new technologies have been introduced. Figure 3 shows a reprint from *"Microservices: the journey so far and challenges ahead"* which gives an overview of some of these technologies [19]. The figure indicates how large and complex the technology stack has become for developing and maintaining your software system.

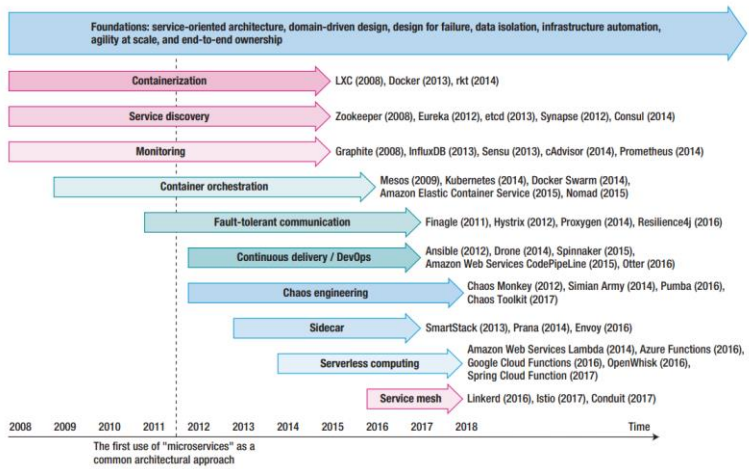


Figure 3 A microservice technologies timeline. Reprint from "Microservices: the journey so far and challenges ahead" [19]

To conclude, autonomous teams are more efficient in a rapid changing environment than teams that must rely upon each other. Within industry the

DevOps methodology is popularized to achieve this team autonomy. DevOps relies on automation practices and distributed system architecture to achieve autonomy. While these practices can be effective, they come at the cost of increased complexity. The right tooling can help managing the increased complexity. This thesis also aims to address the increased complexity with appropriate tooling and focusses on the increased complexity in the documentation of the application landscape.

3.2 Documentation problems

The literature study revealed two main problems with application landscape documentation. The first problem addresses the production process of documentation. And the second problem addresses the effective consumption of this produced documentation. This section will discuss both identified problems.

3.2.1 Documentation production

Several authors have pointed out that Enterprise Architecture documentation[3] is hard to produce and maintain. Roth et al. determined that ~77% (n=108) of EA practitioners either have to apply huge effort to collect data or their data is of poor quality [3]. Studies by Kaisler et al. and Farwick et al. both reported that EA documentation is considered as time consuming, expensive and error-prone [4], [5]. Winter et al. reported that the increasing information volume of organizations combined with the high degree of manual work during the documentation and maintenance of EA models results in a time consuming, expensive and error-prone maintenance of EA information [6].

What these studies have in common is that the observed documentation practices all mainly rely on manual processes. Information for the documentation is gathered through expert interviews by a single or group of solution or enterprise architect(s). And once enough information is gathered the documentation is manually created in static text files with static visualizations. This approach costs much effort, is error prone and the documentation quickly gets outdated.

Automated documentation

Motivated by the problems of manual documentation, research has been conducted on automated documentation. Buschle et al. have researched a technique to generate EA diagrams with the use of an Enterprise Service Bus [20]. Holm et al. researched a technique to generate EA diagrams with the use of an automated network scanner [21]. And Farwick et al. investigated an (semi)automated process for maintaining enterprise architecture models by gathering information from both humans as from live systems [22]. While the studies take an interesting approach, their focus is just on the technical possibilities for extracting documentation. No attention is spent on organizational and usability factors. The studies take a bottom up approach and only investigate the possibilities with certain technology, no attention is spent on what practitioners require.

[3] Enterprise architecture documentation includes the documentation of the application landscape.

To guide the research domain a study by Hauder et al. presented a set of challenges for automated enterprise architecture [23]. They identified four major challenge categories: *data, transformation, business and organization* and *tooling.* The data challenges address difficulties with data quality and the selection of the correct information sources. Transformation challenges include the mapping from information sources to a central repository and the maintenance of this repository. Business and organizational challenges involve the added value of automation and its impact on the structure of the organization. The tooling challenges include the realization of the EA with automated tooling and integration with existing EA databases.

By synthesizing the discussed automation approaches and the results of Hauder et al., one problem of automated application landscape documentation has become apparent. This problem is the technological variability present in a typical application landscape. Due to this variability it becomes very hard for an automated approach to extract all the relevant information out of each system. Therefore, this technological variability in an application landscape is indicated as the biggest problem for an automated documentation approach.

Model-code gap

Fairbanks observed that architectural diagrams often do not reflect the reality of what is happening in the source code [24]. Where architectural diagrams include abstract concepts like components the source code normally does not, although it is able to. Beyond that, architectural models include intentional elements like design decisions and constraints, those cannot be expressed in source code at all. Fairbanks names the discrepancy found between architecture models and the source code the model-code gap.

Architecture reverse engineering approaches attempt to derive high-level models from the source code of a system. Although this type of model gives an accurate description of the source code, due to the model-code gap they often differ from the models sketched by the engineers. And thus, these reverse engineering approaches do not deliver what is required by the engineers.

To bridge the model-code gap Murphy et al. devised a system which compared the high-level models created by the engineers with a reversed engineered model from the source code [25]. With a mapping defined by the user the system would produce a Reflexion Model indicating all the differences between the two models. This way Murphy et al. attempted to bridge to model-code gap.

3.2.2 Documentation consumption

As pointed out in the previous section, the typical application landscape documentation is created as a static text file with a static visualization. To document the architecture of a large application landscape with static diagrams we either require large diagrams or large documents. Documenting the architecture in an all-encompassing diagram results in a large diagram. Splitting this diagram in smaller diagrams results in numerous diagrams and therefore a large document. Both are not desired when consuming documentation, the next section will point out why not.

First, large diagrams are incomprehensible. In George Millers widely cited paper “The Magical Number Seven, Plus or Minus Two” published in 1956 [26]. He

pointed out that the average number of objects an average number can hold in working memory is seven, give or take two. Unfortunately, in enterprise/software architecture this rule is not always considered, sometimes resulting in a single, heavily overloaded, all-encompassing model. Rozanski and Woods describe such a model as the worst of all worlds. They propose that the architecture description should be split up in several views and perspectives [27]. They define a view as: *"a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders"*. Views describe the structural aspects of a system and perspectives describe its quality aspects. Rozanski and Woods describe a perspective as: *"An architectural perspective is a collection of architectural activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of system's architectural views"*. Although their approach helps to structure the documentation to the needs of its consumers. It does not necessarily help to reduce the size of a diagram. A large application can still require a large diagram within one view. With a static representation the only option is to split the large diagram in several smaller, but this will inevitably result in a large document.

Second, large documents are inconvenient. The second point of the Agile Manifesto states *"Working software over comprehensive documentation"*. A large document describing an entire application landscape is comprehensive documentation and therefore will always come second to working software. This makes it unlikely that the people you want involved formulating the documentation will have sufficient time to do so. And second because large documents are large and manually maintained, they are likely to be cumbersome and outdated which makes it implausible they will be consulted.

3.3 Documentation solutions

3.3.1 Documentation production

In search for a solution to cope with the high level of technological variability in a typical application landscape the domain of Model-Driven Engineering (MDE) has been investigated.

Model-driven engineering

MDE strives to raise the abstraction level in software development by working on models instead of directly on source code. To do so MDE uses Domain-Specific Languages (DSLs), this are languages designed for a specific domain, context or organization which help the people in that domain describe things. Brambilla et al. point out four reasons why software development would benefit from the use of models [28]:

"1. Software artifacts are becoming more and more complex and therefore they need to be discussed at different abstraction levels depending on the profile of the involved stakeholders, phase of the development process, and objectives of the work.

2. Software is more and more pervasive in people's life, and the expectation is that the need for new pieces of software or the evolution of existing ones will be continuously increasing.

3. The job market experiences a continuous shortage of software development skills with respect to job requests.

4. Software development is not a self-standing activity: it often imposes interactions with non- developers (e.g., customers, managers, business stakeholders, etc.) which need some mediation in the description of the technical aspects of development."

Although these benefits appear as a good reason to adopt MDE, a global shift in development practices has yet stayed out. Proper tool support has often been blamed for the lacking adoption of MDE [29]–[32]. But, Whittle et al. argue that this is only partially true, in 2015 they have published an extensive taxonomy about all the tool-related issues affecting the adoption of MDE [33]. They did so by placing tooling within a broader organizational context, their analysis resulted in four broad problem themes; *technical factors, internal organizational factors, external organization factors* and *social factors.* Each of these themes has several categories which have in turn sub-categories. For the complete taxonomy the paper should be consulted. In this section the main observations of each theme will be discussed.

Technical factors, one clear observation by Whittle et al. was that MDE can be effective but it takes effort to make it work. Most of their interviewees where successful with MDE but they had either build their own tools or made extensive customizations to standard tools. This indicates that the tools at that time where more are barrier to success rather than an enabler. Furthermore, the usability of

the tools was often poor, available tools could be very powerful, but it was very difficult for the user to access that power. Moreover, the tools did not consider the way how people think, people had to adapt their thinking to the technology instead of the other way around. This includes a lack of attention to the problem-solving process, the tools only provided adequate support once there was known how to solve a problem, but they did not offer support for reaching that solution in the first place. At last the tools often introduced accidental complexity. This means that in practice to optimize usage of the tool often a lot of extra manual work had to be done.

Internal organizational factors, Whittle et al. observed that at the time there was a strong need for tailoring. This means either the tailoring of the tool to the organization, the tailoring of the organization to the tool or building your own tool that fits the organization naturally. All this required tailoring presented a barrier for the adoption of MDE. Furthermore, there is no structured method for knowing which MDE tools are appropriate for which jobs. There is an organizational risk were one successful MDE project leads to applying the MDE technology at several non-appropriate projects. Moreover, MDE can present a curious paradox, where it once was developed to improve portability. In practice issues with versioning and migration often come up reducing the portability. Finally, the way how DSLs spread and grow through an organization is often not under control, this can lead to unacceptable required maintenance, education and tooling costs.

External organizational factors, expectations about what MDE can deliver within an organization are often not well managed. Vendors promise tooling on a high level of abstraction were in reality the abstraction level of the tool is very close to code. Also, the involved costs of the tooling and the indirect cost of training, process change, and cultural shift can act as a barrier for the adoption of MDE. With regards to certification the use of MDE tooling can have both a positive or negative effect depending on the industry and country the company resides in.

Social factors, Whittle et al. taxonomy points out that in general different organizational roles react differently to the adoption of MDE. Software architects tend to embrace MDE because it puts them in control, they can for example encode their architectural rules which forces developers to follow them. Code gurus tend not to embrace MDE because they are afraid, they lose control. Hobbyist programmers also tend to avoid MDE, they are afraid it risks taking away their creativity (similar like a carpenter would not want to risk building Ikea furniture the rest of his life). Managers react differently to MDE depending on their current context and background. In general, MDE requires a fundamental shift in how people work, this will not always be embraced.

Next to all these factors that hamper the adoption of MDE the collective focus on what the real benefits of MDE are might also be off [34]. MDE originated out of the hands of some very technical developers. From the beginning their focus was on the technological factors of MDE and code generation was seen as the holy grail. Now thirty years later this perception is still omnipresent, but recent research has shown that code generation is actually not the key business driver for adopting MDE [34]. It turns out that the main advantages are in the support that MDE provides in documenting a good software architecture, an activity the technology focussed developers where never very fond off. The focus on code

generation led to a marketing strategy where MDE was framed as a technology that could do the same things faster and cheaper. However, this is not usually enough motivation for companies to risk adopting MDE; rather, companies that adopt MDE do so because it can enable business that otherwise would not be possible [34].

Low-code

Since several years a specific version of MDE named low-code is quickly gaining industrial adoption [35]. A low-code platform is a domain specific, model-driven engineering platform focused on the development of enterprise applications. It provides a set of domain specific languages (DSLs) a developer can use to build enterprise applications. Low-code limits its focus on enterprise applications and thereby is able to deliver a better user experience than earlier MDE platforms.

Because it is possible to develop any enterprise application with a low-code platform. The enterprise can develop its entire application landscape with a low-code platform. Because, every application built by a low-code platform is constructed by the same set of DSLs every application will have the same internal structure. This consistency in structure can potentially overcome the problems of technological variation found in a typical application landscape.

3.3.2 Documentation consumption

The second problem to address regards document consumption. We have identified that static diagrams and text files are not fit for the documentation of an application landscape. In search of a solution the domain of information visualizations has been investigated.

Interactive visualizations

In 1996, Ben Schneiderman published a seminal article titled: "*They Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*" [36]. Schneiderman argued that while a page of information is easy to explore, it becomes harder when the information reaches the size of a book or even a library. Therefore, he argued rapid and high-resolution colour displays should be utilized to present large amounts of information in an orderly and user-controlled way. He discovered a principle that summarizes the many visual design guidelines for designing these interactive visualizations. The principle is known as the *Visual Information Seeking Mantra*:

Overview first, zoom and filter, then details-on-demand

Although the principle is based on the design guidelines available in 1996 it still holds true today and is found in most interactive visualizations [37]. To sort out all the different visualizations Schneiderman included a type by task taxonomy for information visualizations. The taxonomy identifies seven data types on which a visualization could be based, combined with seven tasks user could want to perform.

Data types:

1-dimensional:	Linear data types e.g. text documents, source code.
2-dimensional:	Planar or map data e.g. maps, floorplans, layouts.
3-dimensional:	Real world objects e.g. molecules, buildings.

Temporal:	Data with a time dimension e.g. medical records, project management.
Multi-dimensional:	Data with multiple dimensions, found in most relational and statistical databases.
Tree:	Data with a parent child structure e.g. family trees.
Network:	Data where items need to be linked to n number of other items e.g. trade networks, social interaction networks.

Tasks:

Overview:	Gain an overview of the entire collection.
Zoom:	Zoom in on items of interest
Filter:	Filter out uninteresting items.
Details-on-demand:	Select an item or group and get details when needed.
Relate:	View relationships among items.
History:	Keep a history of actions to support undo, replay, and progressive refinement.
Extract:	Allow extraction of sub-collections and of the query parameters.

The static documentation of an application landscape perfectly fits Schneiderman's description of information that is hard to explore due to its size. Investigating if his approach on information visualization will be beneficial for an application landscape is therefore worthwhile.

Because an application landscape essentially is a network of applications connected through dataflows. The network data type would be the most relevant information structure for an application landscape. According to Lee et al., tasks on a network visualization can be categorized as topology based or attribute based [38]. Topology based tasks include finding adjacent nodes or determining a path between nodes. Attribute based tasks include e.g. searching for all nodes with a certain attribute value or edges without a specific attribute value. The use cases of research questions 5 will address which kind of tasks will be relevant for an interactive network visualization of an application landscape.

3.4 Conclusions

Based on the literature study, research question one and two can now be answered.

RQ1: What are the current problems with documenting an application landscape?

In this chapter we identified two main problems of application landscape documentation. The first problem addresses the production process of the documentation. We observed that the production process is time-consuming, expensive and error prone due to its manual nature. To solve this, several automated documentation studies have been conducted. But the high level of technological variability in a typical application landscape is a big hurdle for automated documentation. The second problem addresses the consumption process of the documentation. We have identified that static diagrams and text files are not fit for the documentation of an application landscape. The next section will address these identified problems.

RQ2: How could the current problems with documenting an application landscape be addressed?

The first problem automated application landscape documentation is currently facing is technological variability. This problem could potentially be addressed by basing the automation on a low-code platform.

The second problem is the usage of static text files and visualizations in application landscape documentation. We have identified these static files as unfit for documenting an application landscape due to the size of the involved information. This problem could potentially be addressed by adopting an interactive visualization approach.

Both solutions will potentially work supplementary to each other. An automated approach would have to deliver a structured metamodel with the application landscape information. Such a metamodel is precisely required when creating an interactive visualization.

CHAPTER 4

The Application Landscape Map Requirements

First this section will present some information about the conducted interviews. Second, the results to research question three will be presented.

RQ3: What are the requirements of an interactive language to support a model-based application landscape?

4.1 Interview information

A total of 18 interviews have been held with 14 different practitioners. Three interviews have been conducted in a group of three, one interviews has be conducted in a group of four and all other interviews have been conducted one on one. Some practitioners have only been interviewed once were others two, three or even four times. The group of practitioners consisted of 4 developers, 4 solution architects, 2 enterprise architects, 2 product managers and 2 directors of a low-code vendor. To structure the interviews a semi-structured interview protocol was used. The protocol followed the same structure as the formulated research questions.

4.2 Results

4.2.1 Views and information themes

RQ3.1: What type of visualizations and on demand information should be used to interactively visualize and analyse an application landscape model for enterprise applications?

The interviews indicated that in practice **little usage is made of textbook architecture frameworks**. Practitioners considered them to be too complex and

extensive to support them effectively in practice. When asked about the practices they did use to visualize an existing application landscape their responses were mixed. Some interviewees mainly used improvised informal drawings to visualize an application landscape. While others argued they did not really bother about visualizing the as-is situation because the landscape was familiar to them and it changed too quickly. These interviewees mainly worked on the to-be landscape diagrams and did so using informal drawings. When asked about how new hires are familiarized with the landscape, they did reckon an accurate overview of the as-is situation would be useful.

One interviewee explained his usage of Simon Browns **C4 model** [39]. This model is an industry approach used to describe the **static structure** of a software **system as a map**. It uses a set of four diagrams on different levels of detail to acts as a map through the system. The notation is simple and intuitive, similar to informal boxes and lines drawings. The interviewee had used it during a project and was positive about the approach of the model. He pointed out that a similar approach could also work on application landscape level.

When the C4 model was discussed with other practitioners they responded positive on the approach but commented that the static structure is not the complete picture a landscape. It was suggested to make a distinction between a design-time and run-time representation of the landscape. This distinction should be made because where the C4 model only had to deal with a single system and therefore a single unit of deployment, a landscape approach would have to deal with multiple units of deployment. Having multiple units of deployment can lead to discrepancies between the design-time and run-time landscape. This can happen due to the configuration freedom engineers have when deploying an application. Therefore, practitioners argued both a **design-time** and a **run-time** visualization would be interesting. Furthermore, they addressed it would be interesting to know how often certain API calls are made in production.

To summarize the results for RQ3.1 with regards to the views, the interviewees pointed out the need for a structural view on the landscape from both a design-time and a run-time perspective.

With respect to the relevant on-demand information three themes emerged from the analysis of the interviews. The first theme regards **integrations**, in all interview's information about the integrations between applications was regarded as one of the most valuable aspects for an application landscape. Alongside it the concern was raised that they should be rendered in a smart way because else the picture would get cluttered due to the amount of integrations going on in a typical application landscape. To address this problem one of the interviewees proposed it would help to just show a single relationship between each application and API. All concrete integrations could get shown on-demand once the relationship would be selected. The second theme that came forward was **data**. On application landscape level practitioners addressed it to be interesting to gain insight in what data is stored in what application. Likewise, integrations, the amount of data entities in an application can be huge. Therefore,

measures should be taken to represent the entities in a meaningful way. The last theme that emerged was about **security**. Mainly architects addressed that security information would be very useful for them to have on a landscape level. It would enable them to easily do compliance checking on certain aspects of the application landscape.

Next to addressing the relevant views and required on-demand information some interviewees also came up with several useful requirements for a protentional tool. First multiple interviewees stressed the need for a **filtering** mechanism. Application landscapes at large organizations can consist of hundreds and sometimes even thousands of applications. Therefore, a filtering mechanism is essential to give a user the power to just render what is relevant to him or her. Secondly there was mentioned that an option to **label** certain elements in the visualization would be of great help. This way a user could determine himself to create groups of elements he/she thinks are meaningful. Subsequently interviewees proposed it would be useful to have the ability to **cluster** elements with the same label to one rendered element, this would enable users to reduce the number of elements in the visualization. At last there was mentioned that a user should be able to **save the view** he created on the landscape by filtering and clustering elements, this way it would be easily retrievable for later usage.

The following three lists summarize the results of RQ3.1.

Views:

- Structural design-time view
- Structural run-time view

On demand information themes:

- Integrations
- Data
- Security

Tool requirements:

- Filtering
- Labelling
- Clustering
- Saving custom views

4.2.2 Application landscape elements

RQ3.2: What landscape elements should be displayed in a relevant view?

Research question 3.1 pointed out the need for a structural view on both the design-time and run-time application landscape. The results of research question 3.2 will present the elements of an application landscape that should be included in these views.

In all cases, the interviewees reported the need for a simple representation for both views. Meaning the view should include a minimal number of different

“boxes” and “lines”. In contrast to for example the ArchiMate modelling language in which a plethora of different elements, lines and arrowheads are included. Interviewees stressed to keep the views as simple and intuitive as possible to keep them accessible to a wide range of organizational roles and not just to trained architects.

The interviews surfaced for the design-time view five relevant elements and for the run-time view three relevant elements. Table 2 gives an overview of the elements per view.

	Design-time view	**Run-time view**
Application	X	X
Module	X	
API	X	X
Consume-relationship	X	X
Internal relationship	X	

Table 2 Elements per view, as mentioned by the interviewees.

The application element is the main building block for both the design-time as run-time view. Where in traditional software development the unit of deployment is freely interpreted by each software engineer. In low-code the application is set as the deployable unit of which cannot be deviated. An application is built up through a set of domain specific languages covering UI, application logic, data and much more. A developer is free to determine himself how he structures a system. He can choose to build a system as one large MDE application or he can choose to build a system as a set of interacting smaller applications. But he will always only be able to work in and deploy application files. Therefore, the application being the unit of deployment is a sensible main building block for the two structural views.

The inside of an application is structured in several modules. A module serves as a folder to group a set of coherent DSL files. Therefore, the module is also one of the main structural elements.

An application can expose APIs and consume APIs of other applications. In essence an exposed API is just a DSL file stored in a module. But interviewees addressed it would be interesting to know what applications are connected to what specific APIs of an application. Therefore, it makes sense to represent the APIs as its own element in a view to make explicit to what APIs applications are connecting.

The consume-relationship represents all connections an application is making to a specific API. Interviewees suggested bundling these connections in a cluster for each application API pair would support the readability of the view. Details about integrations should become accessible on-demand.

The internal relationship is an element which shows the connections between modules within an application. Interviewees addressed the relevance of these connections to gain information about internal application coupling.

The interviewees mentioned only for the design-time view a visualization of the internals of an application would be interesting.

4.2.3 On demand information

RQ3.3: What information should be displayed on demand for each element in a relevant interactive view?

During the interviews there has been extensively discussed what information would be relevant to display per element per view. This section will present the findings of these discussions. Table 3 presents what information should be available on demand for the elements in the design-time map, table 4 does the same for the run-time map. Additional information about each table and property can be found in the specification of the ALM in chapter 5.

	General	Integration	Data	Security
Application				
Application name	X			
Application source	X			
Development line	X			
Revision number	X			
Labels	X			
Location	X			
Consumed operation table		X		
Published operation table		X		
Data table			X	
Data access table				X
Module				
Module name	X			
Labels	X			
Module usage table		X		
Used by module table		X		
Data table			X	
API				
API name	X			
API type	X			
Source application	X			

Version	X			
Labels	X			
Published operation list		X		
Connection protocol				X
Authentication protocol				X
Consume Relationship				
Source application name	X			
Target API name	X			
Consumed operations table		X		
Connection protocol				X
Internal relationship				
Source module	X			
Target module	X			
Integration table		X		

Table 3 Design-time view element information.

	General	**Integration**	**Data**	**Security**
Deployed Application				
Application name	X			
Instance id	X			
Source	X			
Version	X			
Location	X			
Consumed operations table		X		
Published operations table		X		
Deployed API				
API name	X			
Source application	X			
Source app instance id	X			
Source module	X			
Version	X			
API type	X			
API location	X			
Link to generated doc	X			
Published operations table		X		
Configured consume Relationship				
Source application name	X			
Target API name	X			
Target API type	X			
Connection protocol				X
Consumed operation table		X		

Table 4 Run-time view element information.

4.2.4 User groups and stories

RQ3.4: What user groups and user stories should be addressed?

The interviews revealed several user groups for which the automated documentation approach is relevant. In this section these user groups will first be presented followed by a set of user stories that are relevant for these user groups. The user stories are grouped by the information themes identified in RQ3.1. The next chapter proposes a specification for a system based on the identified requirements. To evaluate if all the identified user stories are addressed in this specification, each user story is numbered, in the specification there will be referenced to these user story numbers once they are addressed.

A. New developers

First, the ALM can help new developers during their onboarding process. Because the ALM automatically provides an up to date map of the complete landscape, new developers will always have an up to date map they can use to find their way around in the new environment. This while experienced developers will not be bothered maintaining the map.

B. Experienced developers

Second, experienced developers can use the map to perform analysis on (parts) of the landscape or to easily learn about parts of the landscape they are not familiar with.

C. Solution architects

Third, solution architects can use the map to easily do compliance analysis. This way they can check if everyone is adhering to the agreed architectural principles.

General

1. As a new developer, I want to know what applications exist in my company.
2. As a new developer, I want to know where the file of the application model is stored.
3. As a new developer, I want to know which developers have access to which application in the company.
4. As a new developer, I want to know where I can find the documentation of an API.
5. As a new/experienced developer, I want to have an overview of all elements with a specific label.
6. As a new developer, I want to know of which modules an application is constructed.
7. As an experienced developer, I want to know at which URL a specific application instance in a specific environment is deployed.
8. As an experienced developer, I want to know how much instances of an application are deployed in a specific environment.
9. As an experienced developer, I want to know how a specific instance of an application is configured in a specific environment.

10. As an experienced developer, I want to know the impact on the application landscape when I would deploy another development line/revision.

Integration landscape level

11. As a new developer, I want to know what APIs are published in my company.
12. As a new developer, I want to know the types of the APIs in my company.
13. As a new/experienced developer, I want to know what applications are consuming the services of a specific API.
14. As a new developer, I want to know who has integrated with a specific external API before.
15. As a new/experienced developer, I want to know if the intended usage of an API is internal or public.
16. As a new developer, I want to know the operations an API is publishing.
17. As a new developer, I want to know the operations an application is consuming and publishing through APIs.
18. As a solution architect, I want to know which applications have access to a published operation.
19. As a solution architect, I want to know which user roles have access to a published operation.
20. As a solution architect, I want to know from which modules and files API calls are coming.
21. As a solution architect, I want to know if we are consuming external APIs.

Integration module level

22. As an experienced developer, I want to know how tightly the modules in my application are coupled.
23. As an experienced developer, I want to know how much modules a specific module is using.
24. As an experienced developer, I want to know how much a specific module is used by all other modules.
25. As an experienced developer, I want to know how much a specific module is used by a specific other module.
26. As an experienced developer, I want to know of what type the incoming and outgoing usages in a module are.
27. As an experienced developer, I want to know how much of usages are unique.

Integration run-time map

28. As an experienced developer, I want to know how frequent the operation in a consume relationship is being called during run-time.
29. As an experienced developer, I want to know how frequent the published operations of a specific API instance are called during run-time.

30. As an experienced developer, I want to know how frequent all APIs of a specific application instance are called during run-time.
31. As an experienced developer, I want to know how frequent a specific application is calling operations from a specific API.
32. As an experienced developer, I want to know the average request size of an API call.
33. As an experienced developer, I want to know the average response size of an API call.

Data

34. As an experienced developer, I want to know what data is stored in a specific application.
35. As an experienced developer, I want to know what data is stored in a specific module.
36. As an experienced developer, I want to know the state of an entity stored in an application/module.
37. As an experienced developer, I want to know the original owner is of a copy of an entity.
38. As an experienced developer, I want to know if an entity has relevant documentation.

Security

39. As a solution architect, I want to know what connection protocol is used for each application/API.
40. As a solution architect, I want to know what authentication protocol is used for each API.
41. As a solution architect, I want to know what kind of access a specific user role has to a specific attribute in an entity.

CHAPTER 5

The Application Landscape Map Specification

RQ4: How could an interactive language to support model-based application landscapes be constructed?

To answer research question four, a specification for a system has been formulated. In the specification an application landscape metamodel is proposed that should be automatically filled from a set of low-code applications. To evaluate if it is possible to automate this process an expert interview with a product manager from one of the leading MDE vendors has been conducted. The product manager confirmed the possibility and explained that the application models created with their platform could be easily queried and manipulated through their application model SDK. Furthermore, he confirmed that the application models combined with their online platform contained all the required information of the application landscape metamodel.

5.1 Introduction

This is the specification of the Application Landscape Map (from here on referred to as ALM). In this specification the intention of the ALM is explained, along how it should be constructed and for what purposes it can be used.

5.1.1 Objective

The objective of the AALM is to serve as a simple solution to provide an understanding of an application landscape constructed from MDE applications.

5.1.2 Principles

The AALM aims to achieve this objective by adhering to four principles.

First, the map should be interactive, so the user can explore the landscape as he/she sees fit. The user should be in control of what is rendered and what is not.

Second, the map should be simple to understand and intuitive to use. The learning curve of the map should be minimal to keep it as accessible as possible.

Third, the map should be consistent with its actual source. Everything shown on the map should be possible to trace back to the application models.

Fourth, the map should always be up to date. This means it should be possible to generate the map form the source code/application models.

5.1.3 Scope

Where enterprise architecture modelling languages like ArchiMate cover a broad scope including the business, application and technical domain. This first version of the ALM only focuses on the application domain.

5.1.4 Overview

The ALM is inspired by the C4 model of Simon Brown, it likewise aims to construct a map of a software system. However, the focus of the ALM is different, where the C4 model focusses on building a map for a single application the ALM aims to build a map for landscape of applications.

This first version serves as a starting point from which more initiatives can get included. Possible initiatives can be found in the discussion of this thesis.

5.2 Language Structure

Most architectural languages provide a static representation of the architecture and separate the design of the language from an implementation in a tool. This approach helps to keep the language generic and tool independent. But separating the language from its implementation also means some of the dynamic characteristic software has to offer are not fully utilized. The ALM is structured differently, it does combine the design of the language with the implementation in a tool. Thereby it aims to provide a dynamic representation of the landscape.

In order to provide this dynamic representation, the ALM has a layered structure of metamodels as shown in figure 4.

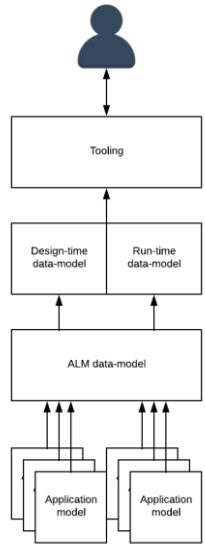


Figure 4 ALM metamodel structure.

First all relevant information from the application models is automatically extracted and stored in the ALM metamodel. Once this model is filled it holds all the information for the entire application landscape, this way it can serve as a central repository for information about the application landscape. The logical version of this metamodel is elaborated in chapter 3.

Second, the ALM metamodel is mapped to several view specific metamodels. These metamodels adhere to a view specific data structure and only store the data relevant for the view. The included views in the specification are a design-time and a run-time view. The metamodels of these views are about what data should be represented for this specific view. Chapter 4 and 5 elaborate on the design-time and run-time metamodels.

Third, a tool can use the view specific metamodels to base a visualization of the landscape on them. This last tooling layer is responsible for how the data is represented in a user interface for each specific view. The tooling layer is about how the data should be represented to the user. Chapter 6 elaborates on the requirements to which a tool implementing the ALM should adhere.

To address the relationships between the entities in the subsequent chapters use has been made of the UML relations notation, figure 5.

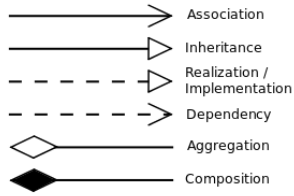


Figure 5 UML relations notation.

5.3 Landscape metamodel

In order to automate the documentation of an application landscape and to provide an interactive visualization, a structured metamodel of the landscape is required. The metamodel of the ALM serves as a central repository for all the required information about the landscape. This metamodel should be automatically filled form the application models. The user of the ALM is not meant to directly interact with the landscape metamodel. To facilitate interaction in the ALM, the landscape metamodel is first mapped to a view specific metamodel which in turn can be used by a user facing tool. This chapter will give the specification for the central landscape metamodel of the ALM. Chapter 4 and 5 will give a specification for two views on this metamodel, the design-time map and the run-time map. The complete metamodel is too large to conveniently print on paper, figure 6. Therefore, it is split up in several parts, each paragraph will discuss one of these parts. Entities that do not belong to a part but are included for reference are made light grey. The metamodel only stores those entities that are relevant to the ALM, it therefore is not a complete metamodel of an application landscape.

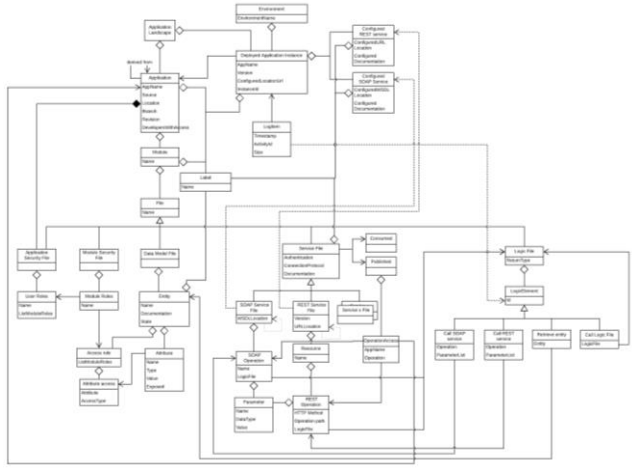


Figure 6 Complete application landscape metamodel.

5.3.1 Application

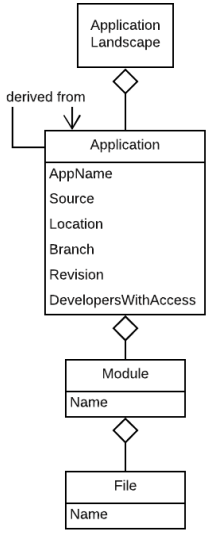


Figure 7 Application landscape metamodel part 1.

Application Landscape

In the ALM metamodel there can only exist one *Application Landscape* entity. It exists on the highest level and everything else is connected to it. The *Application Landscape* can have multiple *Application* entities.

Application

The *Application* is the unit of deployment in the ALM. It is part of a single *Application Landscape* and it consists of one or more *Modules*.

AppName String variable containing the name the user has given to the application.

Source String variable in which the source of the application is stored. This can be “Internal” or “Third-party”, internal means it is part of the MDE landscape to which the ALM should have full access. Third-party means it is an application to which the ALM has minimal or no access.

Location String variable containing the location of an application model. This can be on a local workstation, but also in cloud storage.

Branch String variable containing a specific development branch of an application. For each stored branch a separate *Application* entity should be created.

Revision String variable containing a specific revision of a branch in an application. For each stored revision a separate *Application* entity should be created.

Module

The *Module* is the building block of an application. It enables a developer to make a logical grouping of related files in an application. Except for some best practices there are no technical restrictions to how a developer should structure his application in modules.

Name String variable containing the name of the module.

File

Instead of using code, an MDE application is build up from a set of DSLs (domain specific languages). Each file is structured along the format of a DSL. A domain specific language is a computer language specialized in a specific domain. In the next paragraph all the relevant DSLs for enterprise applications are discussed. A module is built up from a set files that each adhere to the structure of a specific DSLs.

Name String variable containing the name of the File.

5.3.2 File

This paragraph gives an overview of all the Files available to a module. It will also clarify the relation of the files to the domain specific languages. In the consecutive paragraphs each file type will be discussed in more detail.

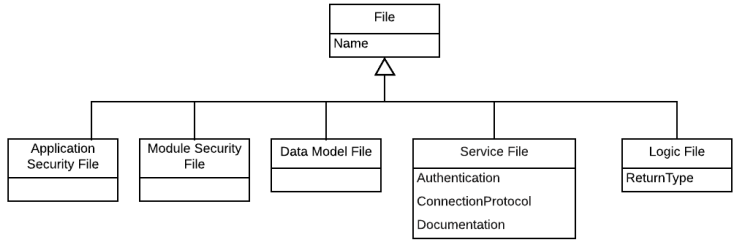


Figure 8 Application landscape metamodel part 2.

Application Security File

File covering security aspects relevant for the complete application.

Module Security File

File covering the security aspects relevant for a module.

Data Model File

File specifying the structure and contents of a data model in an application.

Service File

Set of files that enables the application to consume and publish several types of services.

Authentication String variable describing what authentication protocol is set to access the service.

ConnectionProtocol String variable containing the connection protocol that is required to connect with the service.

Documentation String variable containing documentation added by the user.

Logic File

File that specifies how application logic is handled.

Return Type String variable containing the return type of the logic file.

5.3.3 File relation to DSLs

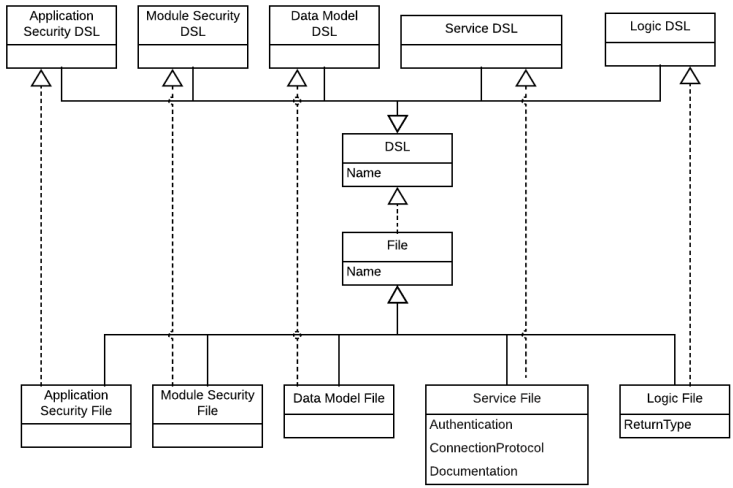


Figure 9 File relationship to DSL.

Each type of file is a realization of a corresponding domain specific language. The DSL specifies the grammar and syntax for the problem domain of the DSL. The files are created by users and follow this DSL grammar and syntax to solve problems specific to each user.

5.3.4 Security and Data File

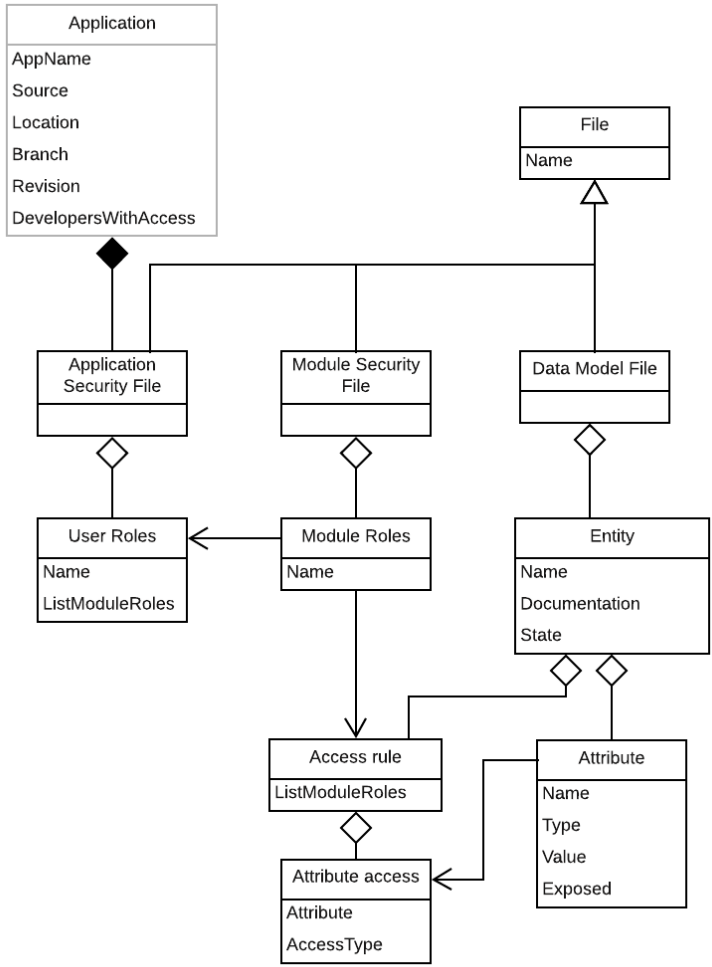


Figure 10 Application landscape metamodel part 3.

Entity

A *Data Model File* consists of a set of *Entities* that store data about certain objects in the application domain. Each *Entity* can have several *Attributes* describing the *Entity* in more detail. For each *Entity* several access rules can get defined.

Name String variable containing the name of the entity.

Documentation Short description of the entity, the developer is free to input any text he wishes.

Attribute

Each *Entity* can have several *Attributes* that store certain aspects of the *Entity.*

Name String value containing the name of the attribute.

Type String value containing the data type of the attribute type.

Value The value to be stored in the in attribute.

Exposed Boolean value storing if the attribute is exposed externally of the application.

Access Rule

An *Access Rule* defines which *Module Roles* have access to which *Entity.*

Attribute Access

Attribute Access defines for each *Module role* in an *Access Rule* the type of access the role has to an attribute.

Module Roles

Module Roles can be assigned to the *Module Security File*. They store different user roles with respect to the specific module. In the *Application Security File,* the modules roles are grouped in *User Roles.*

Name String value containing the name of the module role.

User Roles

In an *Application Security File,* a set of *User Roles* can get defined describing the user of the application. Each *User Role* is a combination of a several *Module Roles.*

5.3.5 Service File

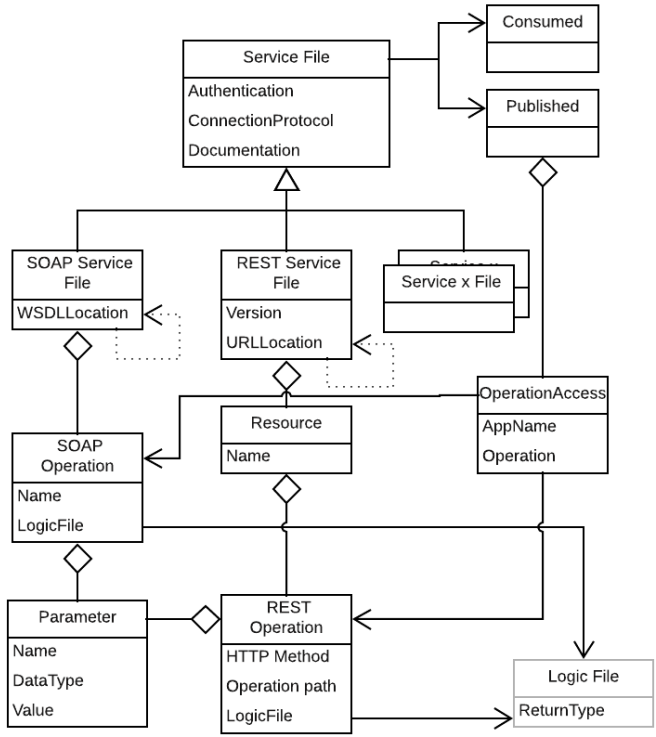


Figure 11 Application landscape metamodel part 4.

Service files can be of two types, either consuming or publishing. A consuming services extract data from an API and publishing services act as an API to expose data to external software.

SOAP Service File

The first supported integration type is a SOAP Service. A *SOAP Service file* consists of a list of several SOAP Operations.

WSDL Location String value containing the location of the WSDL file, this can be an URL or a location path on a specific machine.

REST Service File

The second supported integration pattern is a REST service. A *Rest Service File* consist of a list of several *Resources*.

URL Location String value containing the URL required to access the REST Service

Version String value containing the version number that is intended to be accessed.

Service X

In the current specification of the ALM only SOAP and REST services are included. The Consumed Service X entity represents all the other services that are possible to consume for an application but are not yet included in the metamodel. For example; events, hyperlinks, file imports, SSO integration.

SOAP Operation

Concrete imported operation of the SOAP service, this operation can be called in a logic element.

Name String value containing the name of the SOAP operation.

Logic File String value containing the name of the *Logic File* it is calling.

REST Operation

Concrete imported operation of the REST services, this operation can be called in a logic element.

HTTP Method String value containing the HTTP method of the REST operation, this can for example be; GET, PUT, DELETE etc.

Operation path String value containing the path that need to be added to the *URL location* to access the operation. This path can include parameters.

Logic File String value containing the name of the *Logic File* it is calling.

Resource

Data object which is exposed by the publishing REST Service. A REST service uses the notion of resources to structure several operations on a data object.

Name String value containing the name of the resource/data object.

Parameter

Both SOAP and REST operations can include parameters to make the operations more flexible.

Name String variable containing the name of the parameter.

Data Type String variable containing the data type of the parameter.

Value String variable containing the value of the parameter.

5.3.6 Logic File

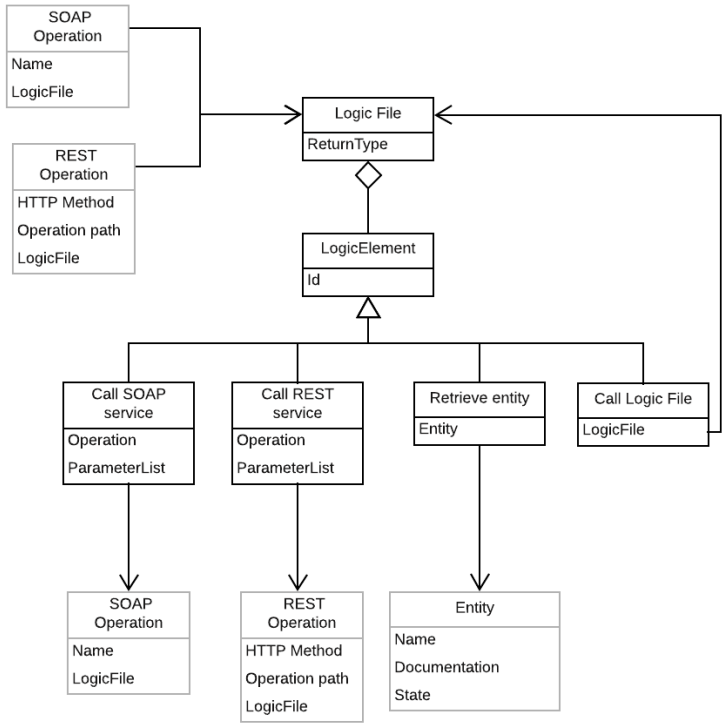


Figure 12 Application landscape metamodel part 5.

Logic Element

A *Logic File* consists of several *Logic Elements* that each perform a specific task. Chaining these *Logic Elements* together in a diagram creates a *Logic File.*

Id String value containing the Id of the *Logic Element.*

Call SOAP Service

The *Call SOAP Service, Logic Element* handles the calls to SOAP services in a logic file. The ALM metamodel only stores the operation the service wants to execute along with its potentially required parameters.

Operation String value containing the name of the operation the developer wants to execute.

Parameter List List of parameters that might be required by the *Operation.*

Call REST Service

The *Call REST Service, Logic Element* handles the calls to REST services in a logic file. The ALM metamodel only stores the operation the service wants to execute along with its potentially required parameters.

Operation String variable containing the complete path required to call the REST operation.

Parameter List List of parameters that might be required by the *Operation.*

Retrieve Entity

The *Retrieve Entity, Logic Element* handles the retrieving of entities from a *Data Model File* to the *Logic File*.

Entity String value containing the name of the *Entity* it wants to retrieve.

Call Logic File

The *Call Logic File, Logic Element* handles the calling of other *Logic Files.* By letting a logic files call other logic files it becomes possible to abstract away from certain parts in a complex logic structure.

Logic File String value containing the name of the *Logic File* it is calling.

5.3.7 Deployment

This part of the metamodel covers the run-time side of an application landscape. Separate entities are used to store the run-time data of the relevant design-time entities.

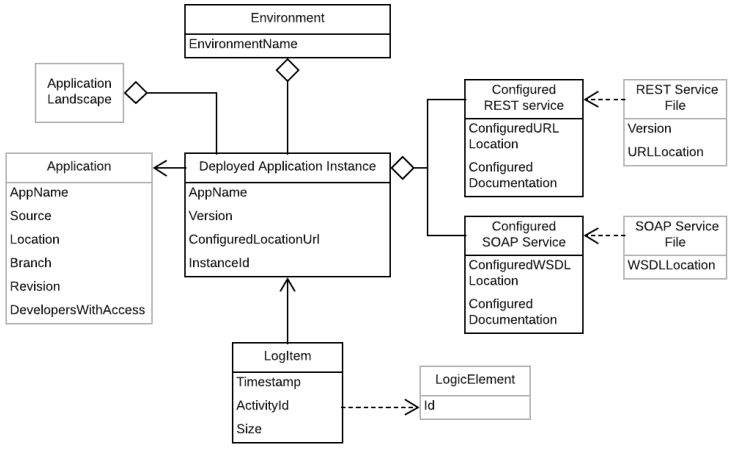


Figure 13 Application landscape metamodel part 6.

Deployed Application Instance

The *Application* entity is the deployable unit in the ALM. Therefore, there also exist *Deployed Application Instance* entities that store all the different deployed instances of an application. The *Deployed Application Instance* consist of several configured services. These configured service entities store the configuration information for the available design-time services.

App Name String value containing the name of the deployed application.

Version String value containing the version of the deployed application, this value is a combination of the Branch and Revision values of the *Application* entity.

Configured Location URL String value containing the URL to access the homepage of the application.

Instance Id String value containing an Id for the specific deployed application. This Id is required because it is possible to deploy the same application with the same version several times.

Environment

A *Deployed Application Instance* is always deployed to a certain *Environment.* The *Environment* entity is used to store all the different environments that exist

in a company. Examples are; test, acceptance, production region 1, production region 2.

Environment Name String value containing the name of the *Environment*.

Configured REST Service

This entity stores the configurations required to deploy a service.

Configured URL Location String value containing the configured URL to access the Service.

Configured REST Service

This entity stores the configurations required to deploy a service.

Configured URL Location String value containing the configured URL to access the Service.

Log Item

For each executed *Logic Element* in a deployed application a *Log Item* entity should be formed. These *Log Items* can be used to analyze an represent runtime behavior of a specific deployed application.

Timestamp Value containing the exact time at which the *Logic Element* was executed.

Activity Id String value containing the Id of the activity that has been executed.

Size Integer value containing the size of the executed operation in bytes.

5.3.8 Labels

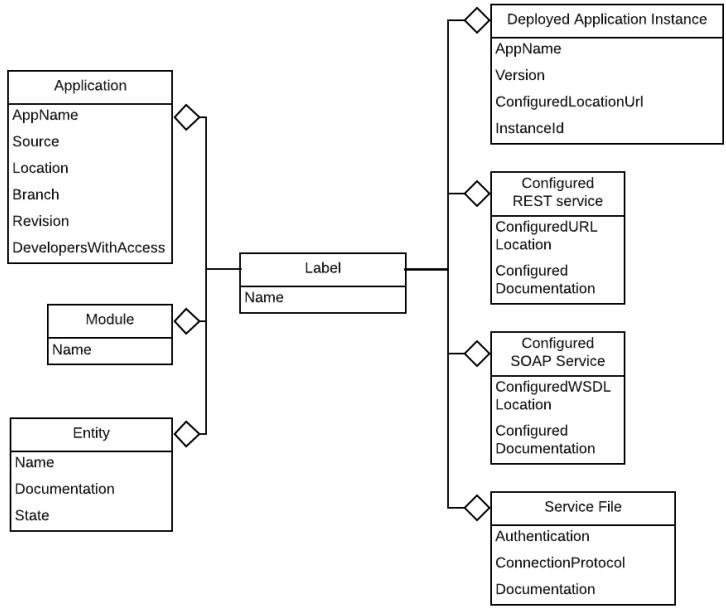


Figure 14 Application landscape metamodel part 7.

Label

Labels can be used freely by a developer to assign several elements in the application landscape to a certain group. Which groups should exist, and which elements should be assigned to these groups is up to the developer.

Name String value containing the name of the *Label*.

5.4 Design-time map metamodel

The design-time map represents the structure of the application landscape during design-time. The basis for the construction of this map is static analysis of application models. It therefore only includes what connections are made in the application model files. How these connections are being utilized once deployed can be found in the run-time map. The map aims to provide a comprehensible interactive representation of the application landscape during design-time.

5.4.1 Metamodel

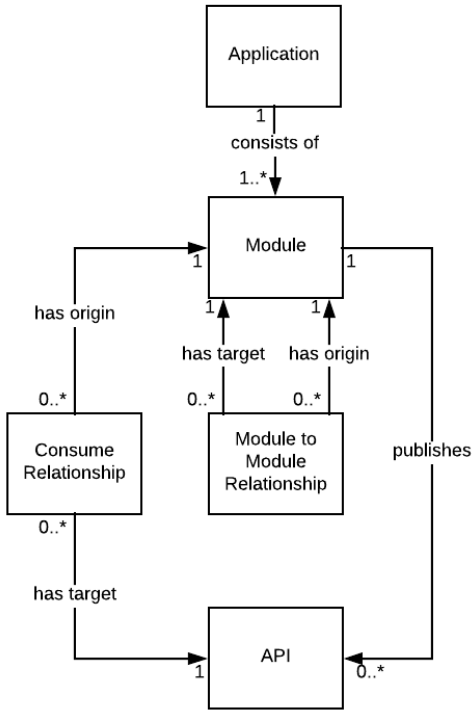


Figure 15 Design-time map metamodel.

The syntax for the design-time map is depicted in the metamodel in figure 15. Each element in the metamodel relates to a structural element that can be found on a rendering of the design-time map. The relations between the elements show the syntax that is allowed.

5.4.2 Element syntax

Application

The application is the main building block of the landscape map. It is an atomic structure of deployment that offers a specific set of functionalities to a

determined user group. An application consists of one or more modules that make up the internal structure of the application.

Module

A module is the internal building block of an application. Modules can have a specific goal determined by the application developer, although this does not have to be the case. The developer has complete freedom how to structure the modules within an application and determine which purpose they should serve. The developer can create as much modules as he/she sees fit.

API

An API provides a structured interface to interact with an application. An API is always published by a single module. An API can be targeted by multiple consumers.

because an application consists of modules an API is part of an application.

Consume Relationship

While an API can be targeted by multiple consume relationships coming from several applications, each individual application can have only one consume relationship to a specific API.

Module-Module Relationship

Modules within an application do not need APIs to integrate with each other. Because they reside in the same application, they can directly interact with one and another. The module to module relationship represents these interactions.

5.4.3 Element properties

Each element of the application map contains a set of properties providing information about the specific element. Tooling should present these properties in a meaningful way to the user. All the element properties should be automatically mapped from the application landscape metamodel to the design-time map metamodel.

Application

AppName String variable containing the name given to the application.

Source Enumeration variable, possible options: “Internal”, “Third-party”. The source property gives information about the origin of the application. The application map is meant to show all applications within control of the user plus the first layer of context to these applications.

Location String variable in which the location of the source code/application model file can be found.

DevelopmentLine Dropdown menu containing all development lines of the application. By selecting a specific development line all application properties and integrations are set to the information found in the source code/application model of that development line.

Revision Dropdown menu containing all revisions of the application. By selecting a specific revision all application properties and integrations should be set to the information found in the source code/application model of that revision.

Labels Array containing all the labels given to the *Application*.

AccesDevelopers String variable containing the names of all developers or developer groups that have access to the application model.

Data Table representing the data stored in the *Application.* The table should include the columns; Module, Domain Model, Entity, Attribute, Ownership, State, User Role, Module Role, Access and Documentation. The first three columns Module, Domain Model and Entity together give the location of an Attribute. The Attribute is the lowest level on which data is stored. The remaining columns give information about the data on Attribute or Entity level. The Ownership column shows on entity level what application is the original owner of the entity. When entities are shared among applications it happens the application that stores the entity is not the original owner. For these shared entities the State column shows what has happened to the imported Entity, for example if it was changed or just enriched. The following three columns; User Role, Module Role and Access, give on an attribute level the access level per User/Module Role. These access levels are: None, Read and Read/Write. The Documentation column shows on an entity level the documentation of the user. It is no problem that this *Data* table element contains a wide range of data columns, it is up to tooling to represent this data in an understandable manner. Chapter 6 will elaborate on this aspect and will give requirements for tooling.

ConsumedOperations Table including all the operations the application is consuming. The table should contain the columns; PublisherApp, PublisherAPI, Operation.

PublishedOperations Table including all the operations the application is publishing. The table should contain the columns; Publishing API, Publishing Operation, Consuming Apps, API location. The columns Publishing API, Published Operation and API location should only contain one value where the Consuming Apps column can contain a list of multiple apps.

Module

ModuleName String variable containing the name given to the module.

Labels Array containing all the labels given to the *Module*.

Data Table representing the data stored in the *Module.* The table should include the columns; Domain Model, Entity, Attribute, Ownership, State, User Role, Module Role, Access and Documentation. The first two columns Domain Model and Entity together give the location of an Attribute. The Attribute is the lowest level on which data is stored. The remaining columns give information about the data on Attribute or Entity level. The Ownership column shows on entity level what application is the original owner of the entity. When entities are shared among applications it happens the application that stores the entity is not the original owner. For these shared entities the State column shows what has

happened to the imported Entity, for example if it was changed or just enriched. The following three columns; User Role, Module Role and Access, give on an attribute level the access level per User/Module Role. These access levels are: None, Read and Read/Write. The Documentation column shows on an entity level the documentation of the user. It is no problem that this *Data* table element contains a wide range of data columns, it is up to tooling to represent this data in an understandable manner. Chapter 6 will elaborate on this aspect and will give requirements for tooling.

UsageOfOtherModules Table showing all the usage the module is making of other modules. The table should contain the columns; IntegrationType, ListIntegratedModules. Each integrated module should be split up in two columns; AbsoluteDocumentIntegrations and UniqueDocumentIntegrations.

UsageByOtherModules Table including all the modules that are making use of this module. The table should contain the columns; IntegrationType, ListIntegratedModules. Each integrated module should be split up in two columns; AbsoluteDocumentIntegrations and UniqueDocumentIntegrations.

API

APIname String variable containing the name given to the API.

Version String variable containing the version of the API. Some APIs also store their version in the location path or name, but because this is not obligatory and consistent for every API the version of an API is also stored in a separate variable.

Type String variable containing the type of the API. Depending on the platform from which the AALM is generated different options are included.

SourceApp Application object referencing to the application which is deploying this API.

SourceModule Module object referencing to the module within an application which is deploying this API.

Documentation Reference to the corresponding automated documentation. For example, to a swagger.io page or a WSDL file.

Source Enumeration variable, possible options: “Internal”, “Public”, “Third-party”. The source property gives information about the origin and intended usage of the API. Internal and Public are both APIs controlled by the organization constructing the ALM. A third-party API is maintained by an external organization over which there is no control. Furthermore, an Internal API is only meant for internal usage. A Public API is meant for usage by external actors to the organization. The application map is meant to show all API within control of the organization plus the first layer of context to these APIs.

ConnectionProtocol String variable containing the connection protocol that is required for connecting to the API. Examples are: http, https.

Authentication String variable containing the authentication protocol that is required to connect to the API.

Labels Array containing all the labels given to the *API*.

PublishedOperations Table including all the operations published by this API. The table should contain the columns; Operation, Consuming Apps, Design-time Access and Run-time Access. The operation column contains all the operations published by this API. The Consumed By column stores per operation which applications are consuming the it. The Design-time Access column stores all the applications that have access to the operation. The Run-time Access column stores which user roles have access to the operation during run-time.

Consume Relationship

SourceApp Application object referencing to the application which is deploying this API.

TargetAPI API object referencing to the API which this consume relationship is targeting.

TypeTargetAPI String variable containing the type of the API which is being targeted.

ConnectionProtocol String variable containing the connection protocol that is being used to connect the SourceApp to the TargetAPI. Examples are: http, https.

ConsumedOperations Table including all the operations the application is consuming. The table should contain the columns; Operation, Module, Document. Each row is used to present a consumed operation, the module and document columns show from which module and document in the application the operation call is being made.

Module-Module Relationship

ModuleA String value containing the name of the module at one end of the relationship.

ModuleB String value containing the name of the module at the other end of the relationship.

Integrations Table showing the integrations between two modules. If the relationship unidirectional the table should just have one column “A->B” if the relationship is bidirectional the table should have another column “B->A”. “A->B” means the usage of ModuleA of ModuleB. Each of these columns should consequently be split up in two columns named; “absolute document usages” and “unique document usages”. In these columns there is shown per integration type how many documents of the other module are being called and how many of them are unique.

5.4.4 Mapping to the metamodel

The following table shows for each attribute in the design-time metamodel where the corresponding data in the complete landscape metamodel can be found.

Application Element		Landscape metamodel
AppName		Application.AppName
Source		Application.Source
Location		Application.Location
Developmentline		Application.Branch
Revision		Application.Revision
Labels		Label.Name
AccessDevelopers		Application.DevelopersWithAccess
DataTable	Module	Module.Name
	Domain Model	File.Name
	Entity	Entity.Name
	Attribute	Attribute.Name
	Ownership	Application.Name
	State	Entity.State
	User Role	User Roles.Name
	Module Role	Module Roles.Name
	Access	Attribute Access.Access type
	Documentation	Entity.Documentation
ConsumedOperationsTable	PublisherApp	Application.Name
	PublisherAPI	File.Name
	OperationList	SOAPOperation.Name, Resource.Name, RESTOperation.HTTP Method, RESTOperation.OperationPath
PublishedOperationsTable	PublishingAPI	File.Name
	PublishingOperation	SOAPOperation.Name, Resource.Name, RESTOperation.HTTP Method , RESTOperation.OperationPath
	ConsumingApps	Application.Name
	APILocation	SOAP Service File.WSDLLocation, REST Service File.URLLocation

Module Element		Landscape metamodel
ModuleName		Module.Name
Labels		Label.Name
DataTable	DomainModel	File.Name
	Entity	Entity.Name
	Attribute	Attribute.Name
	Ownership	Application.Name
	State	Entity.State
	User Role	User Roles.Name
	Module Role	Module Roles.Name
	Access	Attribute Access.Access type

	Documentation	Entity.Documentation
UsageOfOtherModules Table	IntegrationType	Retrieve Entity, Call Logic File
	ListIntegratedModules	Module.Name
	AbsoluteDocumentIntegrations	LogicElement.Id
	UniqueDocumentIntegrations	LogicElement.Id
UsageByOtherModules Table	IntegrationType	Retrieve Entity, Call Logic File
	ListIntegratedModules	Module.name
	AbsoluteDocumentIntegrations	LogicElement.Id
	UniqueDocumentIntegrations	LogicElement.Id

API Element		**Landscape metamodel**
APIName		File.Name
Version		REST Service File.Version
Type		SOAP Service File, REST Service File
SourceApp		Application.Name
SourceModule		Module.Name
Documentation		Service File.Documentation
Source		Application.Source
ConnectionProtocol		Service File.ConnectionProtocol
Authentication		Service File.Authentication
PublishedOperationsTable	Operation	SOAPOperation.Name, Resource.Name, RESTOperation.HTTP Method , RESTOperation.OperationPath
	Consuming Apps	Application.Name
	Design-time Access	Application.DeverlopersWithAccess, OperationAccess.AppName, OperationAccess.Operation
	Run-time Access	Access rule.ListModuleRoles, User Roles.ListModuleRoles

Consume Relationship Element	**Landscape metamodel**
SourceApp	Application.Name
TargetAPI	File.Name
TypeTargetAPI	SOAP Service File, REST Service File
ConnectionProtocol	Service File.ConnectionProtocol
ConsumedOperations	SOAPOperation.Name, Resouce.Name, RESTOperation.HTTPMethod,

		RESTOperation.OperationPath

Module-Module Relationship Element		**Landscape metamodel**
ModuleA		Module.Name
ModuleB		Module.Name
A->B Table	Absolute Document Usages	LogicElement.Id
	Unique Document Usages	LogicElement.Id
B->A Table	Absolute Document Usages	LogicElement.Id
	Unique Document Usages	LogicElement.Id

Table 5 Mapping landscape metamodel to design-time metamodel.

5.5 Run-time map metamodel

The run-time map represents the structure and behavior of the application landscape once it has been deployed. It visualizes a concrete deployed instance of the structure laid out in the design-time map. The metamodel of this run-time map only stores the entities; Deployed Applications, Configured Consume Relationships and Deployed APIs. These three entities are configured, deployed instances of equivalent entities found in the design-time map.

5.5.1 Metamodel

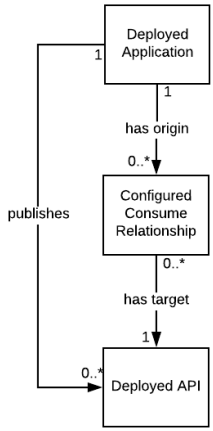


Figure 16 Run-time map metamodel.

5.5.2 Element syntax

Deployed Application

The Deployed Application element is the main building block in the run-time map. Where in the design-time map the application element served as a blueprint for an application. In the run-time map a Deployed Application is a concrete configured instance of this blueprint. Depending on the deployment wishes of the user it is possible that multiple instances of the same application will be deployed. Each instance is its own unique object and can have its own relationships. A Deployed Application always exists in a specific environment which is stored in a property.

Configured Consume Relationship

A Configured Consume Relationship is a configured connection of the Consume Relationship in the design-time map. Not every Consume Relationship in the design-time map has to exist in the run-time map. It depends on the configurations of the user which Consume Relationships exists and how they are configured in each environment. The design-time map determines which application instances **can** get connected to which APIs. The run-time map shows

if this is done and which application instances are connected to which API instances.

While a Deployed API can be targeted by multiple Configured Consume Relationships coming from multiple Deployed Applications, each individual application is only allowed to have one consume relationship to a specific Deployed API. All individual consuming operations are grouped in this Configured Consume Relationship. Having only one relationship between each unique application and unique API reduces the clutter formed in the metamodel of the run-time landscape map.

Deployed API

A Deployed API element provides a structured interface to interact with a Deployed Application. The Deployed API is always part of one Deployed Application. A Deployed API always exists in a specific environment.

5.5.3 Element properties

Deployed Application

AppName String variable containing the name given to the application. This is the same name as used in the design-time map.

Source Enumeration variable, possible options: “Internal”, “Third-party”. The source property gives information about the origin of the application. The application map is meant to show all applications within control of the user plus the first layer of context to these applications.

Location String variable in which the web address of the deployed application is stored.

Version String variable containing the version number of the deployed application instance.

Environment String variable containing the name of the environment to which this application is deployed. Examples are: Test, Acceptance, Production.

Instance String variable containing a unique code for this deployed application instance.

Labels Array containing all the labels given to the *Deployed Application*.

PublishedOperations Table including all the operations the application is publishing along with the frequency they are consumed. The table should contain the columns; Publishing API, Publishing Operation, Consuming Apps, RequestSize, ResponseSize, LastHour, LastDay, LastMonth. The column Publishing API contains the name of the API that is publishing the operation. The column Publishing Operation contains all the operations that are being published by the Application. The column Consuming Apps contains per operation a list of applications that consume the operation. The RequestSize should store the average size of a request of the operation. The ResponseSize should store the average size of the response of the operation. The LastHour,

LastDay and LastMonth columns indicate how frequent the operation has been called during run-time in de corresponding time frames.

ConsumedOperations Table including all the operations the application is consuming along with how frequent it does so. The table should contain the columns; PublisherApp, PublisherAPI, ConsumedOperation, RequestSize, ResponsSize, LastHour, LastDay, LastMonth. The PublisherApp column should store the name of the application that publishes the to consume API. The PublisherAPI column should store the name of the API that publishes the to consume operation. The ConsumedOperation column should store the names of all the operations that are consumed from the API. The RequestSize should store the average size of a request of the operation. The ResponseSize should store the average size of the response of the operation. The LastHour, LastDay and LastMonth columns indicate how frequent the operation has been called during run-time in de corresponding time frames.

Configured Consume Relationship

SourceApp String variable containing the name of the application which is consuming the target API.

TargetAPI String variable containing the name of the API which this consume relationship is targeting.

TypeTargetAPI String variable containing the type of the API which is being targeted.

ConnectionProtocol String variable containing the connection protocol that is being used to connect the SourceApp to the TargetAPI. Examples are: http, https, internal call.

ConsumedOperations Table including all the operations the SourceApp is consuming from the TargetAPI along with how frequent it does so. The table should contain the columns; ConsumedOperation, RequestSize, ResponsSize, LastHour, LastDay, LastMonth. The ConsumedOperation column should store the names of all the operations that are consumed from the API. The RequestSize should store the average size of a request of the operation. The ResponseSize should store the average size of the response of the operation. The LastHour, LastDay and LastMonth columns should store how frequent the operation has been called during run-time in de corresponding time frames.

Deployed API

APIname String variable containing the name given to the deployed API.

Version String variable containing the version of the deployed API.

AppInstance String variable containing the instanceId of its parent application.

Type String variable containing the type of the deployed API. Depending on the platform from which the AALM is generated different options are included.

SourceApp String variable containing the name of the application which is deploying this API.

SourceModule String variable containing the name of the module within an application which is deploying this API.

APILocation String variable with the URL at which the deployed API is accessible.

Documentation Reference to the corresponding automated documentation. For example, to a swagger.io page or a WSDL file.

Source Enumeration variable, possible options: “Internal”, “Public”, “Third-party”. The source property gives information about the origin and intended usage of the API. Internal and Public are both APIs controlled by the organization constructing the ALM. A third-party API is maintained by an external organization over which there is no control. Furthermore, an Internal API is only meant for internal usage. A Public API is meant for usage by external actors to the organization. The application map is meant to show all APIs within control of the organization plus the first layer of context to these APIs.

Labels Array containing all the labels given to the *Deployed API*.

PublishedOperations Table including all the operations the API is publishing along with the frequency they are consumed. The table should contain the columns; Publishing Operation, Consuming Apps, RequestSize, ResponseSize, LastHour, LastDay, LastMonth. The column Publishing Operation contains all the operations that are being published by the API. The column Consuming Apps contains per operation a list of applications that consume the operation. The RequestSize shows the average size of a request of the operation. The ResponseSize show the average size of the response of the operation. The LastHour, LastDay and LastMonth columns indicate how frequent the operation has been called in de corresponding time frames.

5.5.4 Mapping to the metamodel

The following table shows for each attribute in the run-time metamodel where the data in the complete landscape metamodel can be found.

Deployed Application		Landscape metamodel
AppName		Application.AppName
Source		Application.Source
Location		Deployed Application Instance.ConfiguredLocationUrl
Version		Deployed Application Instance.Version
Environment		Environment.Environment Name
Instance		Deployed Application Instance.InstanceId
Labels		Label.Name
ConsumedOperationsTable	PublisherApp	Deployed Application Instance.InstanceId
	PublisherAPI	File.Name
	OperationList	SOAPOperation.Name, Resource.Name, RESTOperation.HTTP Method , RESTOperation.Operation Path
	RequestSize	LogItem.ActivityId, LogItem.Size
	ResponseSize	LogItem.ActivityId, LogItem.Size
	LastHour	LogItem.ActivityId, LogItem.Timestamp
	LastDay	LogItem.ActivityId, LogItem.Timestamp
	LastMonth	LogItem.ActivityId, LogItem.Timestamp
PublishedOperationsTable	PublishingAPI	File.Name
	PublishingOperation	SOAPOperation.Name, Resource.Name, RESTOperation.HTTP Method , RESTOperation.Operation Path
	ConsumingApps	Deployed Application Instance.InstanceId
	RequestSize	LogItem.ActivityId, LogItem.Size
	ResponseSize	LogItem.ActivityId, LogItem.Size
	LastHour	LogItem.ActivityId, LogItem.Timestamp
	LastDay	LogItem.ActivityId, LogItem.Timestamp

	LastMonth	LogItem.ActivityId, LogItem.Timestamp

Configured Consume Relationship		**Landscape metamodel**
SourceApp		Deployed Application Instance.InstanceId
TargetAPI		File.Name, Deployed Application Instance.InstanceId
TypeTargetAPI		SOAP Service File, REST Service File
ConnectionProtocol		Service File.ConnectionProtocol
ConsumedOperationsTable	ConsumedOperation	SOAPOperation.Name, Resource.Name, RESTOperation.HTTP Method , RESTOperation.Operation Path
	RequestSize	LogItem.ActivityId, LogItem.Size
	ResponseSize	LogItem.ActivityId, LogItem.Size
	LastHour	LogItem.ActivityId, LogItem.Timestamp
	LastDay	LogItem.ActivityId, LogItem.Timestamp
	LastMonth	LogItem.ActivityId, LogItem.Timestamp

Deployed API		**Landscape metamodel**
APIname		File.Name
Version		REST Service File.Version
AppInstance		Deployed Application Instance.InstanceId
Type		SOAP Service File, REST Service File
SourceApp		Deployed Application Instance.InstanceId
SourceModule		Module.Name
APILocation		Configured SOAP service.ConfiguredWSDLLocation, Configured REST Service.ConfiguredURLLocation
Documentation		Configured SOAP service.ConfiguredDocumentation, Configured REST Service.ConfiguredDocumentation
Source		Application.Source
Labels		Label.Name

PublishedOperationsTable	PublishingOperation	SOAPOperation.Name, Resource.Name, RESTOperation.HTTP Method, RESTOperation.Operation Path
	ConsumingApps	Deployed Application Instance.InstanceId
	RequestSize	LogItem.ActivityId, LogItem.Size
	ResponseSize	LogItem.ActivityId, LogItem.Size
	LastHour	LogItem.ActivityId, LogItem.Timestamp
	LastDay	LogItem.ActivityId, LogItem.Timestamp
	LastMonth	LogItem.ActivityId, LogItem.Timestamp

Table 6 Mapping landscape metamodel to run-time metamodel

5.6 Tool requirements

Tooling plays an integral part in the ALM, all the metamodels are set-up to accommodate maximal freedom for a tool. This chapter will outline a set of requirements and propose several designs for a potential tool. The designs are based on the user stories described in section 4.2.4. The red numbers within the figures act as a reference to specific user stories. A number indicates the user story is addressed by the information near the number. When a number is placed in the bottom right corner it means the content of the entire figure acts as a solution to the user story.

5.6.1 Filtering Elements

In large domains the complete set of applications in the design-time or run-time map can become very large, this presents a risk of the canvas getting cluttered with dots and lines. To avoid this from happening a filter mechanism should be available to the user. This mechanism should allow the user to filter out each element shown on the canvas. Filtered elements will remain to exist, but they will not be shown on the canvas for the moment. Such a mechanism will allow the user to determine himself what is important to show and what not. Additionally, it should be possible to quickly filter a set of elements which all hold the same Label.

5.6.2 Clustering Elements

To reduce the potential complexity of a landscape the tool should be able to cluster a set of elements. Once clustered the set of elements should be rendered as a single element with a special color or symbol indicating it is a group. When the user clicks on the group element the context window should show what elements it consists of. The user should be given complete freedom over how to arrange these clusters. Additionally, it should be possible to quickly cluster a set of elements which all hold the same Label.

5.6.3 Context window

The tool should provide both the design- and run-time map with a context window. This context window should show the properties of the selected element by the user.

5.6.4 User views

By filtering elements form the landscape and by grouping certain elements the user is able to create his own user view on the landscape. The user should be able to save this user view, so he can easily access it again. Because the landscape map is automatically updated, and user views are created manually they can get outdated. Therefore, once an outdated user view is loaded the tool should give an update of all changes to the landscape since the last save.

5.6.5 Visualization of the design-time map

The design-time map should be able to show two different zoom levels. The first zoom level should focus on the complete application landscape and its context of external applications. The second zoom level should just focus on the internals of one application and its context in the landscape.

Design-time landscape zoom

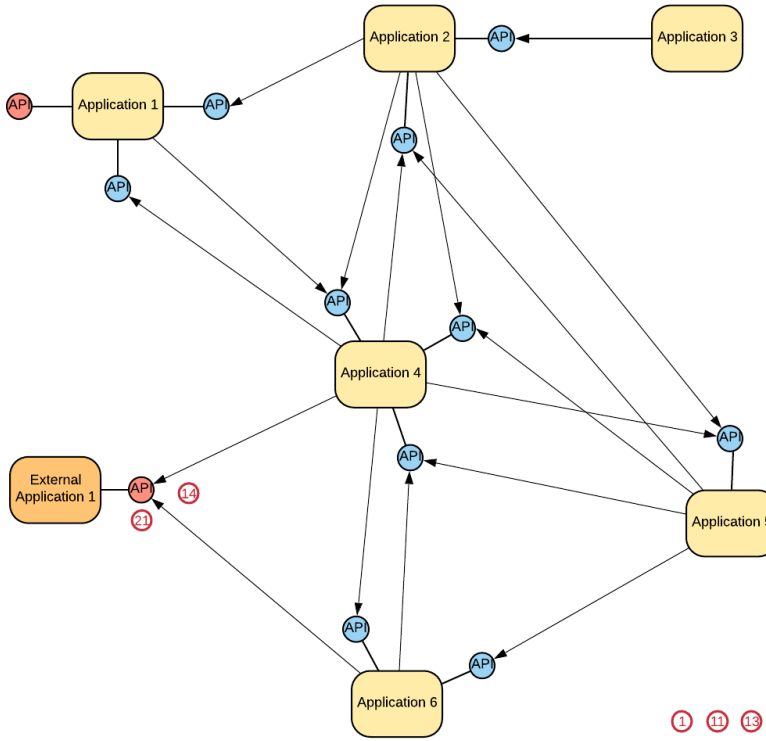


Figure 17 Design-time landscape map visualization

Figure 17 above shows how a design-time application landscape could get presented in a graph view. The color of the applications indicates if they are internal (light yellow) or external (orange). An application is internal when it is controlled by the company for which the landscape is created. External applications are controlled by another company of which the company has no control. The color of the APIs indicates if the API is internal (blue), public (red) or third-party(orange). An internal API is only meant for communication between internal applications. A public API is meant to be consumed by external applications. An third-party API is similar to an external application, it is controlled by an external company and consumed by the landscape. The directions of the arrows indicate consume relationships. The tail is connected to

the application that is consuming, and the head is connected to the API that is consumed. All the aspects described in section 6.1-6.4 should be applied to the landscape graph view. By double clicking on an application element the view should shift to the *design-time application zoom*.

Design-time application zoom

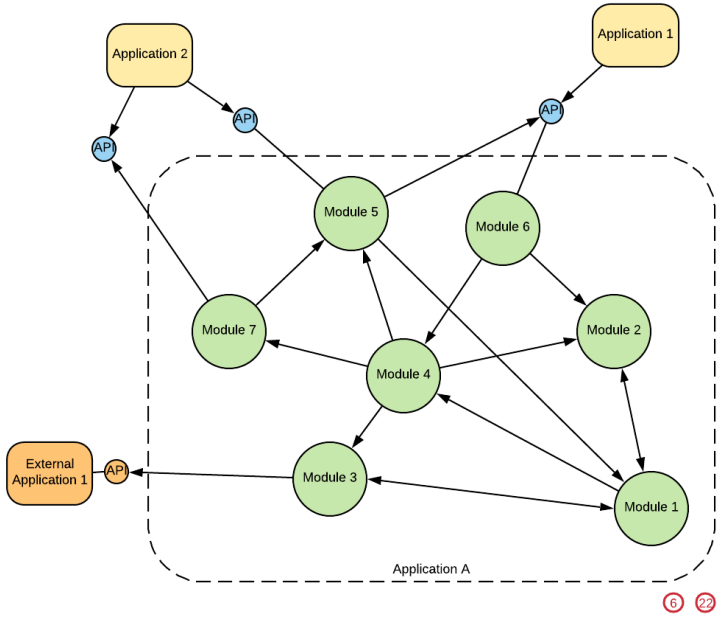


Figure 18 Design-time single application map visualization.

Figure 18 above shows how the internals of a single application could get visualized in a graph view. The view shows the individual modules that make up the application and how they are connected to each other. Furthermore, it shows the first layer of context to the application. The arrows represent direct usage relationships because internally in an application it is not required to communicate through APIs (although this is possible as well). Because the arrows indicate direct usage relationships, unlike the landscape view they can be bidirectional. The aspects described in section 6.1-6.4 should be applied to this view as well.

Context window design

Once an element is selected in a view its properties should get displayed in a context window. This section presents how this information should be visualized.

Application

Once the user selects an application on the design-time map the context window should show its attributes as shown in figure 19.

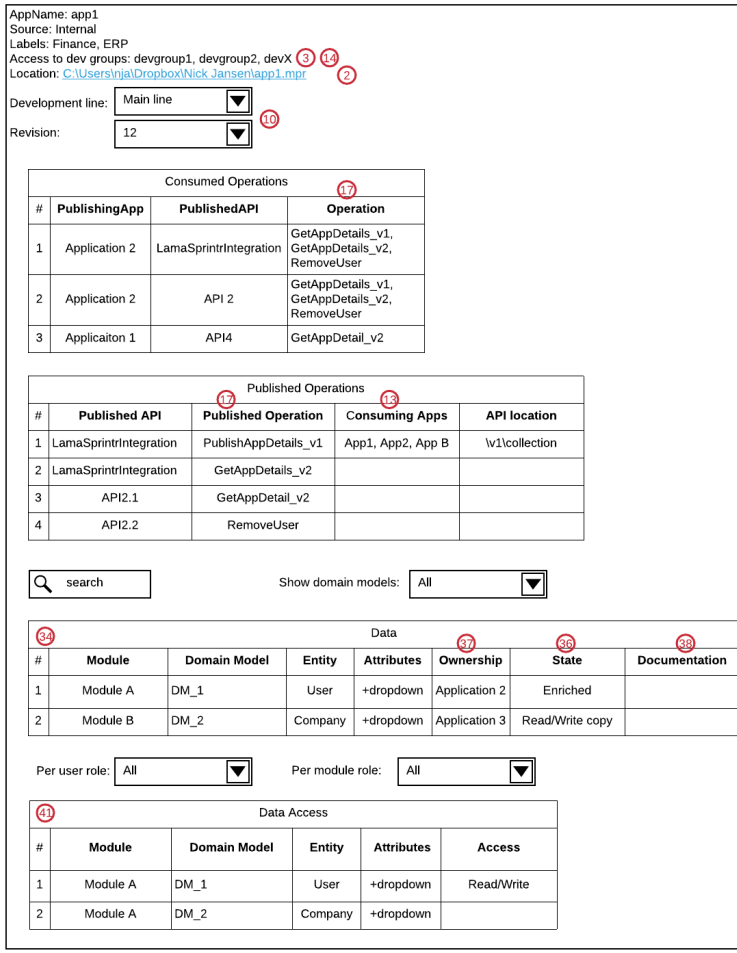


Figure 19 Application context window design.

In the Consumed Operations table all operations should be grouped per PublishedAPI. This way a PublishingApp can have multiple rows for several PublishedAPIs. But a PublishedAPI just contains one row for all consumed operations. Grouping them will this way will reduce the size of the table.

Module

Once the user selects a module on the design-time map the context window should be visualized as shown in figure 20.

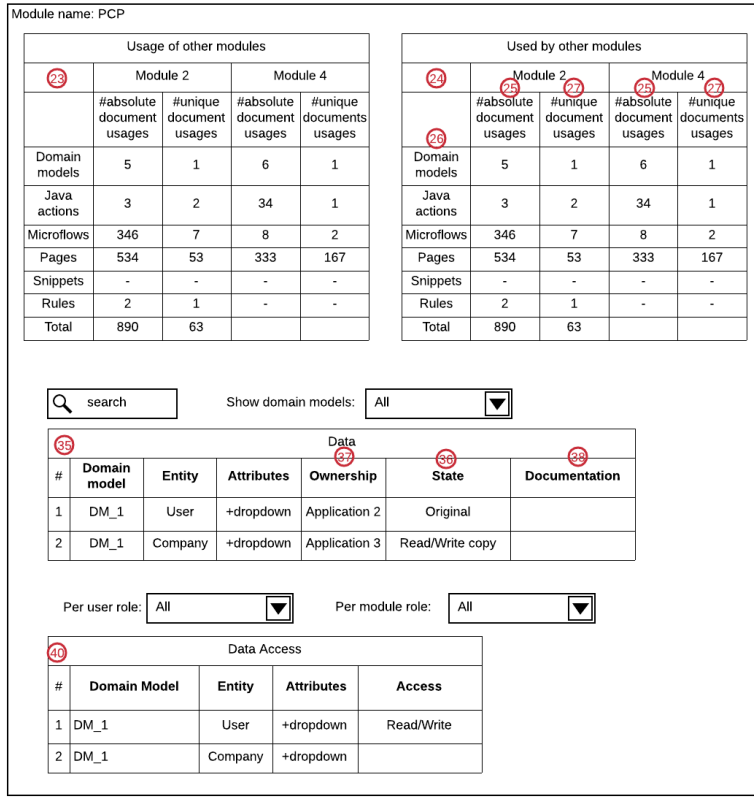


Figure 20 Module context window design.

API

Once the user selects an API on the design-time map the context window should be visualized as shown in figure 21.

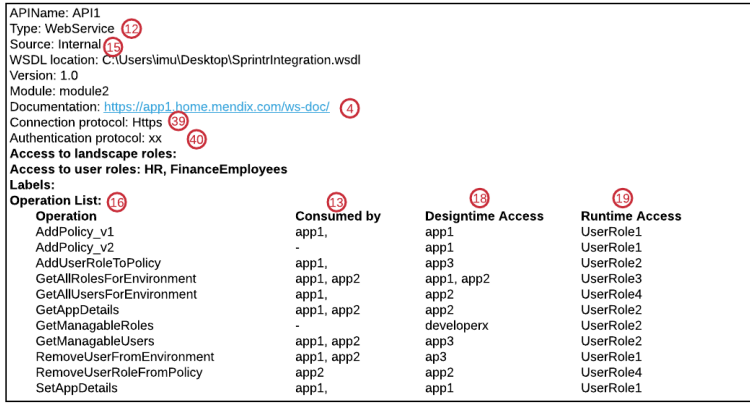


Figure 21 API context window design.

Consume relationships

The origin of a consume relationship lays in a module, but because an application consists of modules the application can also be used as the origin in the visualization for the landscape zoom. A consume relationship shows direction to the to be consumed API. Once the user selects an consume relationship on the design-time map the context window should be visualized as shown in figure 22.

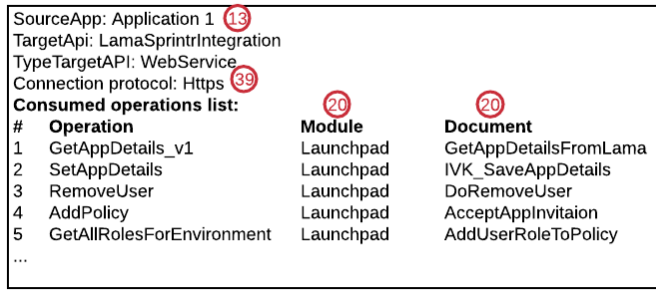


Figure 22 Consume relationship context window design.

Module-Module relationship

Each unique module can only have one relationship to another unique module. Allowing just one relationship between each module pair reduces the clutter formed in the rendering of the landscape map. A module-module relationship can be bi-directional. When this is the case the context window should show information of the integrations in both directions. Once the user selects a Module-Module relationship on the design-time map the context window should be visualized as shown in figure 23.

ModuleA: Module1
ModuleB: Module3

	A -> B (25)	(27)	B -> A (25)	(27)
(26)	#absolute document usages	#unique documents used	#absolute document usages	#unique documents used
Domain models	5	1	6	1
Java actions	3	2	34	1
Microflows	346	7	8	2
Pages	534	53	333	167
Snippets	-	-	-	-
Rules	2	1	-	-
Total	890	63		

Figure 23 Module-module relationship context window design.

5.6.6 Visualization of the run-time map

The run-time map only has to include the zoom level on which the complete application landscape and its context are shown. The zoom level that focusses on a single application is not relevant during run-time.

Because generally an application landscape is deployed to several environments for testing purposes, multiple run-time maps should be generated, one map for each unique environment. The user should be able to easily switch between these different maps.

Run-time landscape zoom

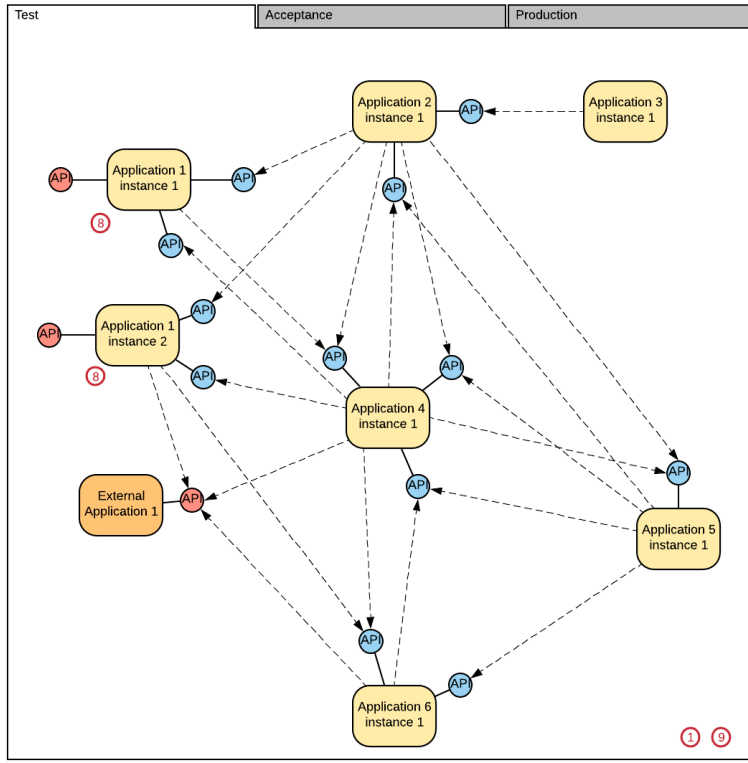


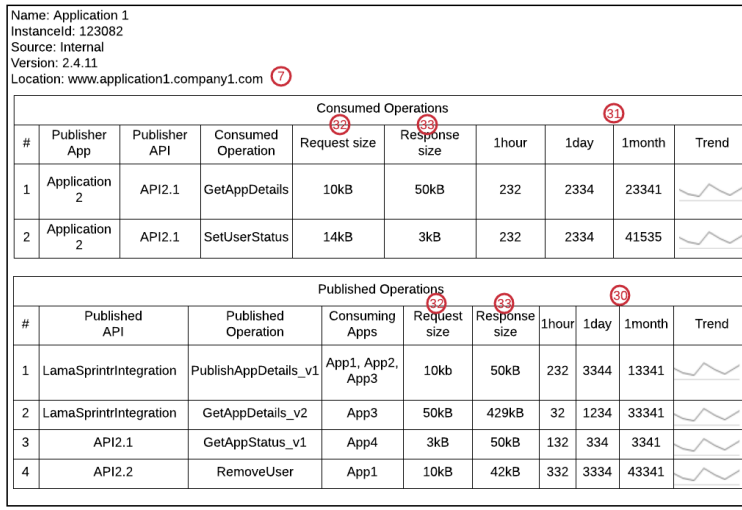
Figure 24 Run-time map landscape visualization

Figure 24 shows a graph view for the run-time landscape. It is similar to the view for the design-time landscape, but except of applications it shows application instances. As exemplified in the figure it is possible to deploy multiple instances of an application and configure both differently. The consume relationships are dashed to indicate they represent the flow of run-time data. Furthermore, tabs are included for each deployment environment, clicking on the tabs should let the user easily switch between the environments.

Context window design

Deployed Application

Once the user selects a deployed application on the run-time map the context window should be visualized as shown in figure 25.


Name: Application 1
InstanceId: 123082
Source: Internal
Version: 2.4.11
Location: www.application1.company1.com (7)

Consumed Operations (31)

#	Publisher App	Publisher API	Consumed Operation	Request size (32)	Response size (33)	1hour	1day	1month	Trend
1	Application 2	API2.1	GetAppDetails	10kB	50kB	232	2334	23341	
2	Application 2	API2.1	SetUserStatus	14kB	3kB	232	2334	41535	

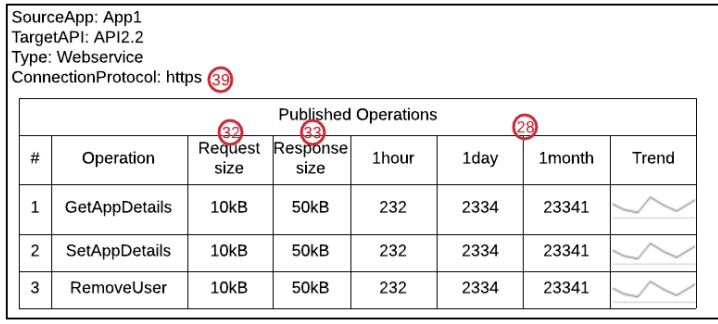
Published Operations (30)

#	Published API	Published Operation	Consuming Apps	Request size (32)	Response size (33)	1hour	1day	1month	Trend
1	LamaSprintrIntegration	PublishAppDetails_v1	App1, App2, App3	10kb	50kB	232	3344	13341	
2	LamaSprintrIntegration	GetAppDetails_v2	App3	50kB	429kB	32	1234	33341	
3	API2.1	GetAppStatus_v1	App4	3kB	50kB	132	334	3341	
4	API2.2	RemoveUser	App1	10kB	42kB	332	3334	43341	

Figure 25 Deployed application context window design.

Configured Consume Relationship

Once the user selects a configured consume relationship on the run-time map the context window should be visualized as shown in figure 26.


SourceApp: App1
TargetAPI: API2.2
Type: Webservice
ConnectionProtocol: https (39)

Published Operations (28)

#	Operation	Request size (32)	Response size (33)	1hour	1day	1month	Trend
1	GetAppDetails	10kB	50kB	232	2334	23341	
2	SetAppDetails	10kB	50kB	232	2334	23341	
3	RemoveUser	10kB	50kB	232	2334	23341	

Figure 26 Configured consume relationship context window design.

Deployed API

Once the user selects a deployed API on the run-time map the context window should be visualized as shown in figure 27.

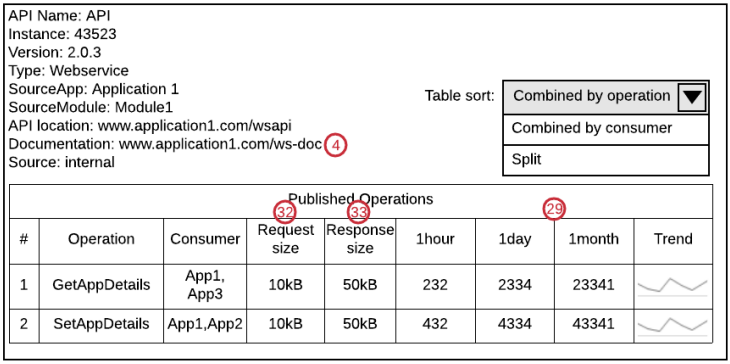


Figure 27 Deployed API context window design.

5.7 Proof of concept

To evaluate how a tool could look and behave, a proof of concept has been constructed based on the specification in this document. In the current version the landscape and application zoom levels are combined in one view, later versions could potentially split this in two different views. The proof of concept uses demo data and is not yet linked to application models of an MDE platform.

The proof of concept can be found at:

http://nick-jansen.nl/ALM-Graph.html

The proof of concept only stores demo data for the context window behind elements indicated with an *.

CHAPTER 7

Initial Evaluation

An initial evaluation of the proposed specification has taken place during its design. By adopting the ADSRM the specification has been built and evaluated iteratively. During each interview practitioners have been asked how they thought about the current design and how it could be improved. By using this approach, it was possible to evaluate the design often and early in the process, guiding the design process with practitioner feedback. Once the main design cycle was completed all interviewees where contacted once more to ask them about the final design. The majority of the interviewees responded positive to the final design. One interviewee addressed that the current specification only focusses on the as-is situation of the application landscape where the interviewee would be more interested in support for the to-be situation. This limitation along with several others will be elaborated in the discussion chapter.

Furthermore, the design has been evaluated by demonstrating that each user story is addressed in the specification. The interviews resulted in a large set of user stories presented in chapter 4.2.2. Each user story is linked to the part the design that addresses it in chapter 5.6. This way there has been evaluated if the design conforms to the wishes of the users.

Although most interviewees reported that the proposed tool would greatly help them, we have not been able to prove if it would be an improvement over a traditional application landscape production/consumption approach. To empirically validate if our approach is an improvement. First an implementation of the ALM specification should be developed. And second, a large-scale experiment would have to be set up to test our automated approach against a traditional approach. This study serves as a good starting point for such an evaluation.

Major changes during iterative evaluation

The ALM specification was constructed in numerous iterations. During these iterations several large changes and many small changes have been implemented.

This section will elaborate on the large design changes and explain their motivations.

Relationship visualization

The visualization of the relationships/dependencies between different applications is an important aspect of the ALM. Before the final design was reached several other designs have been proposed and evaluated.

One of the first designs included an arrow notation for each dependency between two applications. This approach has been tested on a section of a large low-code application landscape, see figure 28.

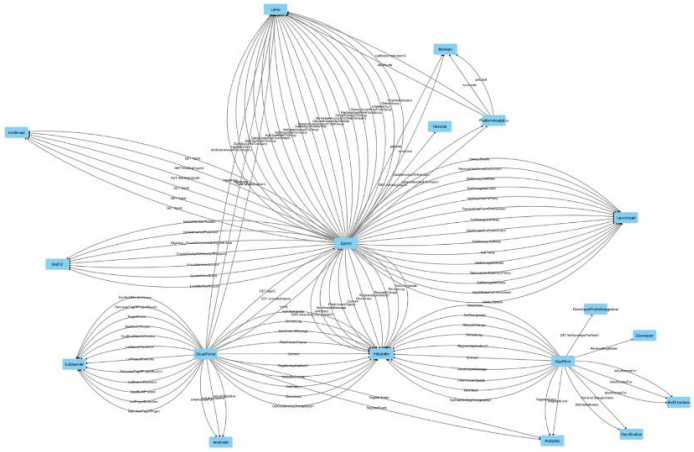


Figure 28 ALM design all dependencies visualized.

The evaluation of this design concluded that although complete, the level of detail was to high, resulting in a complex cluttered view on the landscape. Therefore, several other versions have been designed to explore how interactive elements could reduce the complexity, see figure 29.

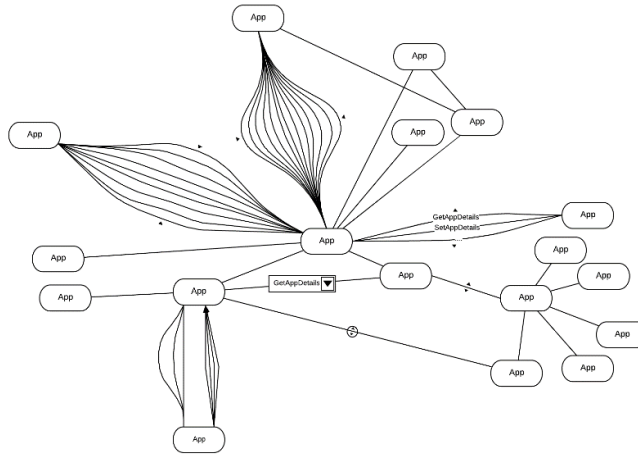


Figure 29 ALM folding mechanisms study.

Figure 29 shows several options how relationships between applications could be hidden or exposed. It shows a folding mechanism which could hide/show individual dependencies and a dropdown menu which hides all arrows and lets the user select a specific one from a list. After evaluating these designs there was concluded that although the complexity in amount of lines was partially reduced. Partially because when an arrow would be “unfolded” still a large amount of lines would remain, making it difficult to pinpoint a specific dependency. The complexity also increased by adding more interface elements.

These conclusions led to the final design where only one arrow per application-API pair is allowed, by selecting the arrow the details of all the individual dependencies are shown in a context window, figure 30. This way the cluttering of the view is reduced, and the details become available on demand in a specific window.

Figure 30 ALM one arrow per application-API

API visualization

The evaluation of the design shown in figure 30 was positive about the way the individual dependencies are hidden. But also pointed out the developers need to specifically know which APIs are consumed by which applications.

Therefore, in the next design the APIs are included as stand-alone elements, figure 31. This way it becomes immediately clear which APIs each application is consuming and publishing. Besides, it becomes easier to find information about specific APIs by simply selecting them and looking in the context window. In the visualization the APIs should remain in the vicinity of the application that is exposing them.

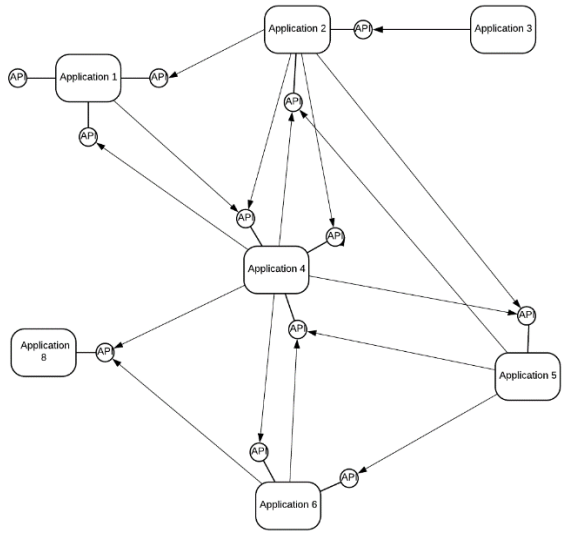


Figure 31 ALM API visualization

Design time, run time map

As discussed in the results chapter the interviewees pointed out the need for a separate view on the run time application landscape. Due to the freedom software engineers have when configuring and deploying an application, the run time landscape can differ from the design time landscape.

Interviewing the software architects revealed that in some cases the same set of applications is deployed numerous times for e.g. different regions. Such a use case can result in a large run time landscape with a lot of repetition. To address this use case, first a design was proposed that included a tab for each repetitive section, which can for example be a region, figure 32. But while evaluating this approach with the architects there was concluded that it would be confusing to include these repetitive sections in separate tabs because it could be understood

as if they exist in separate environments. This is not the case because each of these repetitive sections is connected to a single (large) production environment. Based on this feedback there was decided it would be a better approach to use the labeling mechanism for these kinds of repetitive sections. Because they system should allow to cluster elements based on a label, clustering could also be done for these different regions if they are labeled correctly. Such an approach will reduce the size and complexity of run time landscapes including deployments for different regions.

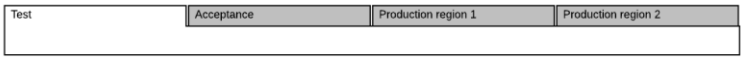


Figure 32 A tab for each region.

CHAPTER 8

Discussion & Conclusions

This study set the first steps in exploring how application landscape documentation could be generated from low-code applications. During the study several areas of interest emerged that revealed limitations of the current approach but also highlighted opportunities for future work. Because they could not be included or addressed in the current study they will be discussed in this chapter.

8.1 Limitations

First this section will discuss the limitations of the current study. The subsequent section will address these limitations by proposing directions for future research.

L1: Limited evaluation

The current study is yet unable to prove if the proposed solution is an improvement over a traditional documentation approach. Although the design of the solution has been literately validated through interviews with practitioners and their response has been positive. An empirical comparison between the proposed solution and a traditional approach has yet to be conducted.

L2: Data population proof of concept

In the current proof of concept, the data in the data model is hardcoded to investigate the visualization of the landscape. Although the possibility to automatically fill the metamodel from low-code applications has been confirmed through an expert interview. The implementation of this automation has yet to be developed and evaluated.

L3: Visualization methods

Landesberger et al. defined three main groups for visualizing graphs: *node-link based*, *matrix-based* and *hybrid* [40]. To keep a realistic scope this study just focused on node-link visualizations. But throughout the study there was realized, that a matrix-based visualization could have several advantages over node-link visualization. Based on the proposed metamodel it would be relatively easily to

include other visualizations. But as of yet the specification is limited to just a node-link visualization.

L4: Inclusion of non-low code applications

The strength of the automated documentation approach defined in this study is also its weakness. Because the approach limits itself to low-code applications it becomes achievable to create a valuable automated data model of an application landscape. But in a typical enterprise the application landscape does not solely consist of low code applications. In this study the design of the proposed solution is limited to low-code applications, future research could investigate how to include non-low code applications.

8.2 Directions for future research

This section will elaborate on several interesting directions future research could take based on the results of this study.

Empirical evaluation (L1)

Because the initial responses to the proposed design have been positive. It would be interesting for future research to empirically investigate of the proposed solution is an improvement over a traditional documentation approach. First the proposed specification of this study would have to be implemented. And second a comparison study could be set up to empirically validate if the automated interactive documentation approach is an improvement over a manual static documentation approach. By conducting such a study, the main research question of this study could be answered with certainty.

Data population proof of concept (L2)

Future work could focus on a proof of concept that includes automatic population of the metamodel based on low-code applications. Several cases studies based on different low-code vendors can be conducted to hopefully a find universal approach for all low-code applications.

Visualization methods (L3)

This study has scoped itself to node-link visualizations for the application landscape. But while conducted the research we came across some use cases for which the node-link visualization might not be the best fit. When one wants to analyze the structure of a large set of elements the node-link visualization does not scale well. Although a filtering mechanism was included in the design of the interactive node-link visualization, in some situations a practitioner might want to analyze the complete set of elements. For these cases a matrix visualization might be helpful. The main advantage of a matrix view is that it scales much better than a node-link visualization. Where at some point a node-link visualization turns into an incomprehensible cloud of lines and arrows, a matrix stays clean and comprehensible. The main disadvantage is the increased difficulty to follow paths between the analyzed elements. Figure 33 presents an example of a matrix view, visualizing the relationships between all modules in a large low-code application.

Figure 33 Matrix visualization of a large low-code application.

In a matrix each module is put both on the x- and the y-axis, the cells indicate if there exists a relationship between the module on the x- and y-axis of the cell. The matrix reserves 2 cells for each relationship between the two modules. One cell at the upper half of the diagonal and one cell at the lower half (the diagonal itself shows where each module meets itself). Symmetric matrices indicate there just exists a relationship between two modules. Asymmetric matrices also give the direction of the relationship. In the asymmetric matrix in figure 33 a colored cell indicates the module on the y-axis is used by the module in the x-axis. Unlike a general relationship in a symmetric matrix, a used by relationship naturally does not have to go in both directions, therefore the matrix becomes asymmetric.

The ordering of the rows and columns plays an important role in a matrix visualization. The order of rows and columns can be freely modified as long as the same happens on both axes. Different ordering strategies exist, including ordering by frequency, cluster, name and layer. These ordering strategies can be a powerful tool to analyze a complex application/landscape. Ordering by cluster can for example reveal several sets of tightly coupled modules in a large application. When faced with the refactoring of a monolithic application to microservices this can be valuable information. Furthermore, ordering by layer can reveal cyclic dependencies that indicates software complexity, this information can be input for refactoring projects.

Based on the metamodel proposed in this thesis it would be possible to construct a matrix visualization next to a graph visualization. The matrix visualization could be used for 1) a single application indicating the relationships between its modules. 2) an application landscape indicating the relationships between its applications. Or 3) an application landscape with all application modules exposed, this way we can analyze the relationships between modules residing in different applications.

Due to the large amount of information an application/landscape matrix could potentially visualize, an interactive approach would be interesting to investigate. Based on the knowledge gained knowledge in this study we propose the following interactive controls for the matrix visualization, figure 34. The controls should be able to do the following:

Dependency direction:	Select the direction of relationships represented in the matrix visualization.
Order:	Select the ordering algorithm of the matrix, different ordering algorithms facilitate different types of analysis.
Included dependencies:	Select the type of dependencies that should be included in the matrix. In a low-code application, numerous different dependencies between modules exist, the user should be able to select which one he would like to include in his analysis.
Opacity:	Set the opacity of the cells to visualize how strong a relationship between two modules is. By allowing a user to set his own low and high limit, he is in control of what he considers a strong or weak relationship.
Selected cell:	Context window displaying general information for the at the time selected cell in the matrix.

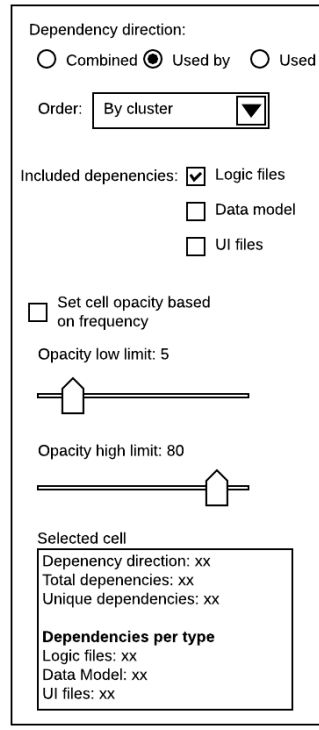


Figure 34 Proposed interactive controls matrix visualization

Based on a large low-code application a proof of concept has been constructed. It was built to experiment with an interactive visualization and different ordering mechanisms. The proof of concept can be found at:

http://nick-jansen.nl/ALM-DSM.html

Inclusion of non-low code applications (L4)

As discussed at the limitations, the current solution only includes low-code applications. Unfortunately, the typical application landscape contains a lot more variation, therefore it would be interesting to investigate how non-low-code applications could be included in the metamodel. The usages of stubs or network scanners could potentially provide solutions.

Model modification

This study focused its scope on the reconstruction and analysis of the current application landscape. An interesting next step would be to investigate if it would help practitioners to make alterations on the abstraction level of an application landscape. And if so, how these changes should be pushed back to the individual application models. Figure 35 illustrates all the steps, step 1 and 2 has been the focus of this study, step 3 lays open for future research.

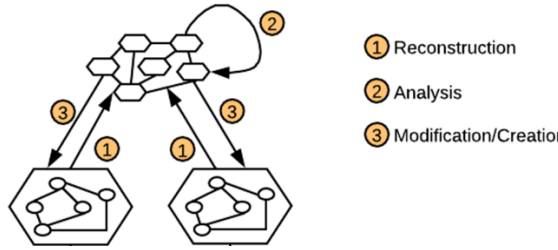


Figure 35 Application landscape activities.

Evolutionary architecture

Evolutionary architecture as proposed by Ford et al. is defined as [41]:

"An evolutionary architecture supports guided, incremental change across multiple dimensions."

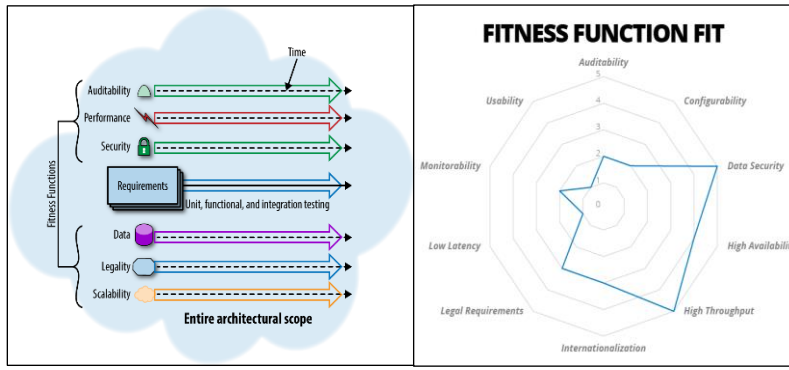


Figure 36 Evolutionary architecture fitness functions. Reprint from "Building evolutionary architectures : support constant change" [41].

Ford et al. propose that the relevant quality attributes of a software architecture should be continuously measured through a defined protocol, also called fitness functions, figure 36 illustrates this concept. This way after each commit engineers could receive feedback on the impact their changes have on the software architecture. Making them aware of the potentially architecture eroding effects of their changes. This way they can be guided at each increment to make decisions in line with the agreed upon architecture. The hard part of implementing evolutionary architecture is defining these fitness functions.

Future research could investigate if an application landscape metamodel as proposed in this study could be and interesting source of information for defining these fitness functions.

8.3 Conclusions

This thesis set out to answer the following question:

***MQ**: How could the documentation of an application landscape be improved?*

A literature study first identified two potential problems with current application landscape documentation. First, we observed that due to its manual nature the current production process of application landscape documentation is time-consuming, expensive and error prone. To solve this, several automated documentation studies have been conducted. But the high level of technological variability in a typical application landscape is a big hurdle for these automated documentation approaches. Second, we have identified that static diagrams and text files are not fit for the documentation of an application landscape.

Consequently, the same literature study found two potential solutions to these identified problems. The first problem of technological variability can potentially be addressed by basing automation on a low-code platform. And the second problem of static text files and visualizations being unfit for application landscape documentation could potentially be addressed by adopting an interactive visualization approach.

To test these findings there has been investigated what practitioners would require of a documentation approach based on low-code automation and interactive visualizations. The ADSRM was chosen as a research method and by conducting a series of interviews the requirements for a system have been formulated.

Based on these requirements a case study has been conducted. This case study led to the specification of the Application Landscape Map. The ALM illustrates how a documentation approach based on low-code automation and interactive visualizations could look like. This specification attempts to answer the main research question. At last a discussion elaborates on the limitations and future directions of the proposed system. Because there only has been conducted a case study at a single company its generalizability is still questionably. But because the initial response has been positive this study serves as a promising starting point for future research.

References

[1] N. Dragoni *et al.*, “Microservices: yesterday, today, and tomorrow,” 2017.

[2] M. Lankhorst, *Enterprise architcture at work: Modelling, communication an analysis.*, Third Edit. Springer Science & Business Media, 2009.

[3] S. Roth, M. Hauder, M. Farwick, R. Breu, and F. Matthes, “Enterprise architecture documentation: Current practices and future directions,” *Wi*, no. 2013, pp. 912–925, 2013.

[4] M. Farwick, R. Breu, B. Agreiter, 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. SECTISSIMO View project Software Quality Assurance View project REQUIREMENTS FOR AUTOMATED ENTERPRISE ARCH,” 2011.

[5] S. H. Kaisler, F. Armour, and M. Valivullah, “Enterprise Architecting: Critical Problems,” in *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, 2007, vol. 37, no. 4, p. 224b–224b.

[6] K. Winter *et al.*, “Association for Information Systems INVESTIGATING THE STATE-OF-THE-ART IN ENTERPRISE ARCHITECTURE MANAGEMENT METHODS IN LITERATURE AND PRACTICE Recommended Citation "INVESTIGATING THE STATE-OF-THE-ART IN ENTERPRISE ARCHITECTURE MANAGEMENT METHODS IN ,” 2010.

[7] H. W. J. Rittel and M. M. Webber, “Dilemmas in a General Theory of Planning*,” 1973.

[8] J. Conklin, “Social Complexity Wicked Problems,” 2001.

[9] K. Conboy, R. Gleasure, and E. Cullina, “LNCS 9073 - Agile Design Science Research,” *LNCS*, vol. 9073, pp. 168–180, 2015.

[10] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A Design Science Research Methodology for Information Systems Research.”

[11] M. Conway, “How do committees invent?,” *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[12] A. Maccormack, J. Rusnak, and C. Baldwin, “Exploring the Duality between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis,” 2007.

[13] F. W. Taylor, “Scientific Management,” *Sociol. Rev.*, vol. a7, no. 3, pp. 266–269, Jul. 1914.

[14] M. Fowler, “The State of Agile Software in 2018,” 2018. [Online]. Available: https://martinfowler.com/articles/agile-aus-2018.html. [Accessed: 21-Jan-2019].

[15] R. Jeffries, “Dark Scrum,” 2016. [Online]. Available: https://ronjeffries.com/articles/016-09ff/defense/. [Accessed: 21-Jan-2019].

[16] L. Bass, I. M. Weber, and L. Zhu, *DevOps : a software architect’s perspective*. .

[17] S. Newman, *Building Microservices*. 2015.

[18] E. Evans, *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley, 2004.

[19] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, 2018.

[20] M. Buschle, S. Grunow, F. Matthes, M. Ekstedt, M. Hauder, and S. Roth, “Automating Enterprise Architecture Documentation using an Enterprise Service Bus,” *Am. Conf. Inf. Syst. (AMCIS 2012)*, pp. 1–14, 2012.

[21] H. Holm, M. Buschle, R. Lagerström, and M. Ekstedt, “Automatic data

collection for enterprise architecture models," *Softw. Syst. Model.*, vol. 13, no. 2, pp. 825–841, 2014.

[22] M. Farwick, B. Agreiter, R. Breu, S. Ryll, K. Voges, and I. Hanschke, "Automation processes for enterprise architecture management," *Proc. - IEEE Int. Enterp. Distrib. Object Comput. Work. EDOC*, no. October, pp. 340–349, 2011.

[23] M. Hauder, F. Matthes, and S. Roth, "Challenges for automated enterprise architecture documentation," *Lect. Notes Bus. Inf. Process.*, vol. 131 LNBIP, pp. 21–39, 2012.

[24] G. Fairbanks, *Just Enough Software Architecture*. 2010.

[25] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software Reflexion Models: Bridging the gap between Source and High Level Models," *IEEE Trans. Softw. Eng.*, vol. 27, no. 4, pp. 364–380, 2001.

[26] G. A. Miller, ""The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing..," vol. (Mar). 1963.

[27] J. Brownell, "Opening the 'Taj': The Culture of Fantasy," *Cornell Hotel Restaur. Adm. Q.*, vol. 31, no. 2, pp. 19–23, 1990.

[28] N. Rozanski and E. Woods, *Software systems architecture : working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.

[29] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven software engineering in practice*. 2012.

[30] J. Den Haan, "8 reasons why model-driven approaches (will) fail," 2008. [Online]. Available: https://www.infoq.com/articles/8-reasons-why-MDE-fails.

[31] A. Kuhn, G. C. Murphy, and C. A. Thompson, "An Exploratory Study of Forces and Frictions affecting Large-Scale Model-Driven Development."

[32] M. Staron, "Adopting Model Driven Software Development in Industry-A Case Study at Two Companies Software Center View project Adopting Model Driven Software Development in Industry-A Case Study at Two Companies."

[33] F. Tomassetti, A. Tiso, F. Ricca, M. Torchiano, and G. Reggio, “POLITECNICO DI TORINO Repository ISTITUZIONALE Maturity of Software Modelling and Model Driven Engineering: a Survey in the Italian Industry / Maturity of Software Modelling and Model Driven Engineering: a Survey in the Italian Industry Maturity of Software Modelling and Model Driven Engineering: a Survey in the Italian Industry,” no. Spain, pp. 14–15, 2012.

[34] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, “A taxonomy of tool-related issues affecting the adoption of model-driven engineering,” *Softw. Syst. Model.*, vol. 16, no. 2, pp. 313–331, 2015.

[35] J. Whittle, J. Hutchinson, and M. Rouncefield, “The State of Practice in Model-Driven Engineering,” *Software, IEEE*, vol. 31, no. 3, pp. 79–85, 2014.

[36] J. Rymer, C. Mines, A. Vizgaitis, and A. Reese, “The Forrester Wave™: Low-Code Development Platforms For AD&D Pros, Q4 2017.” [Online]. Available: https://reprints.forrester.com/#/assets/2/225/’RES137262’/reports. [Accessed: 22-Jan-2019].

[37] B. Shneiderman, “The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations,” 1996.

[38] Scott Murray, *Interactive Data Visualization for the Web: An Introduction to Designing with - Scott Murray - □□□ Google*. 2017.

[39] B. Lee, C. Plaisant, C. Sims, J.-D. Fekete, N. Henry, and C. Sims Parr, “Task taxonomy for graph visualization,” pp. 1–5, 2006.

[40] S. Brown, “The C4 model for software architecture.” [Online]. Available: https://c4model.com/. [Accessed: 24-Jan-2019].

[41] T. Von Landesberger *et al.*, “Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges,” vol. 30, no. 6, pp. 1–28, 2011.

[42] N. Ford, R. Parsons, and P. Kua, *Building evolutionary architectures : support constant change*. O’Reilly Media, 2017.

[43] F. P. Brooks, “The Mythical Man-Month.” 1975.