Unification of Organizational Data Processing for Inner Source

MASTER THESIS

Philipp Uriel Winklmann

Submitted on 1 March 2024

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Stefan Buchner, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 1 March 2024

License

Erlangen, 1 March 2024

Abstract

This thesis addresses the challenge of managing organisational structure data in inner-source software development environments and proposes a novel unified structure for standardised data storage and processing. The development of this unified structure and an implementation approach aims to solve identified problems, with demonstrations and evaluations showing potential solutions and highlighting limitations.

Our work contributes to academic and practical areas by solving the critical problem of organising enterprise data and increasing the efficiency of data management practices. Despite the progress, the complexity of organisational data management shows that further research is needed, especially regarding system integration and adaptability.

Future directions include exploring different use cases that may require alternative design decisions and extending the unified structure into a comprehensive employee knowledge base. This could open up new applications, such as detailed employee evaluations and process analyses, previously hindered by scattered data in different systems.

This work lays the foundation for a standardised concept of organisational data management and thus represents a significant development for the future.

Contents

List of Figures

List of Tables

Acronyms

IS Inner Source

OS Open Source

AD Active Directory

ERP Enterprise Resource Planning

HCM SAP Human Capital Management

HR SAP Human Resources

CRM Customer Relationship Management

$O(x)$ Complexity of runtime or storage according to the Big O notation

$O(n)$ Linear complexity according to the Big O notation with dependence on number of entities in an organisation, i.e. employees and departments

$O(m)$ Linear complexity according to the Big O notation with dependence on number of employees in an organisation

$O(d)$ Linear complexity according to the Big O notation with dependence on number of departments in an organisation

$O(E)$ Linear complexity according to the Big O notation with dependence on number of connections between departments and departments and between departments and employees in an organisation

1 Introduction

1.1 Thesis Motivation

Inner Source (IS) is a software development approach that adapts open-source development methodologies to organisations, allowing all developers of one company to contribute to projects across different departments or teams while maintaining a corporate setting (Capraro & Riehle, 2016). This collaborative approach has been shown to increase code reuse, promote knowledge sharing, and improve software quality (Carroll et al., 2018), (Cooper & Stol, 2018).

However, with any software development, there are many departments, such as finance, HR and management, who have to know how long programming tasks have taken in the past and how long they might take in the future. In addition, Stol et al. (2014) underline the greater importance of coordination and leadership in IS projects. These tasks also require a solid information base including an overview of current contributions for their day-to-day business.

As Buchner (2022) lays out, calculating the contribution of individual developers is even more complex once IS is introduced in a company. The reason is that developers in IS often work on many different projects, such that today's standard solutions of time tracking or checking in and out of projects are no longer feasible. Therefore, a solution is needed to automatically track a developer's contribution to a specific code base or project. In addition, this contribution of individual developers then needs to be added up per department and IS project for tax reasons.

As a solution to this, Buchner (2022) also presents a method to determine the time a developer spends on an individual code contribution. Other than that, there is a contribution by Gurbani et al. (2010) that discusses managing corporate Open Source (OS) software assets by tracking developer contributions and licenses used.

One problem that both articles face is the question of how to effectively determine which developer is assigned to which department, control unit or similar unit of interest.

In this thesis, we therefore present a novel method to determine a developer's place in an organisation's hierarchical structure using organisational structure data. This identified gap serves as the driving impetus behind this research. Our work aims to overcome this challenge by developing a unified standard for organisational data storage that can accommodate data from different systems yet be flexible enough to integrate with existing tools.

Given these insights, the following research question arises: "What can a unified standard for organisational data storage in IS environments look like?"

1.2 Structure of this Thesis

The thesis follows the design science approach by Peffers et al. (2007) and is structured into eight chapters, each addressing a specific aspect of the proposed solution. An overview of the chapters can be seen in figure 1.1.

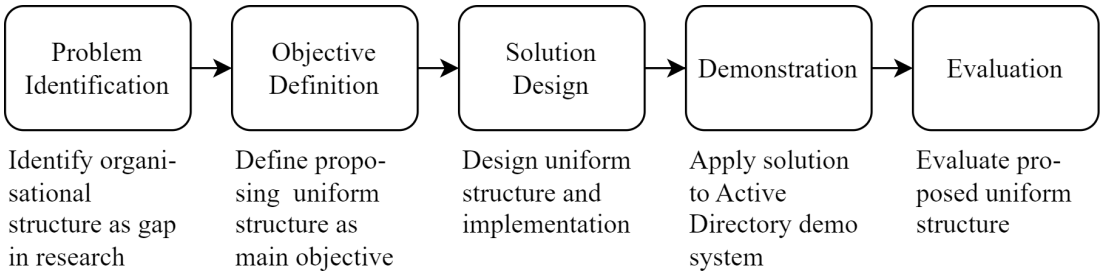

Figure 1.1: Application of design science steps by Peffers et al. (2007) to this thesis

Chapter 1 provides an introduction to the thesis and its objectives.

Chapter 2 identifies the problem this thesis aims to solve, namely identifying the organisational structure of an organisation to determine which developer works in which department. This is necessary for tax and management purposes in organisations using IS and derives the need for a uniform structure to unify organisational structure data from different systems.

Chapter 3 defines the objectives to the identified problems, which is to propose a uniform structure and an implementation to map developers to departments.

Chapter 4 outlines the design of the proposed solution, which involves the development of a uniform structure for organisational structure data and the implementation of this structure to unify structure data and map developers to departments.

Chapter 5 describes the implementation of the proposed solution, which involves

applying the algorithms to extract and unify the required data and map developers to departments.

Chapter 6 demonstrates the effectiveness of the proposed solution by presenting an exemplary implementation for an Active Directory demonstration system.

Chapter 7 evaluates the proposed solution and checks whether the set goals have been met.

Chapter 8 concludes the thesis by summarising the key findings, discussing the strengths and weaknesses of the proposed solution and suggesting areas for future improvement.

2 Problem Identification

As outlined in the thesis motivation, businesses face the need to track individual developers' contributions to software projects and aggregate these contributions at various organisational levels - departmental as well as project levels. Research proposes two methods for keeping track of developers' contributions to projects (Buchner, 2022), (Gurbani et al., 2010). Both fall short regarding aggregating this information at various organisational levels as needed for tax and budgeting reasons. Examples of these organisational levels are departments, divisions, and sections.

The reasons for this gap are the inability to link developers to their departments and the lack of knowledge about the organisations' internal structure to aggregate the data further. Both are necessary to gain the ability to analyse and predict the contributions of different organisational levels.

The data on the organisational structure must be used to gain the necessary knowledge about the internal structure of the organisations. This data refers to the information that outlines the affiliation of employees and the hierarchy within a company or institution. Organisational structure data needs to be present in every company for different reasons. The simplest is that every company is required by law to appoint at least one CEO. Aside from that, organisational structure data is often used to assign permissions to IT accounts, providing an overview of the organisation or the previously mentioned budgeting and tax reasons.

The only problem is that this information is presented in a wide variety of forms and qualities. In order to be able to use the information contained, the structure and quality of the data available in individual cases must be evaluated first. Only then could the function of mapping a developer's contribution to a specific organisational level be included in existing tools.

The underlying problem is that there is no unified structure for organisational structure data available that could be implemented into existing tools to solve the other identified problems.

3 Objective Definition

The main objective of this master thesis is to pursue a set of interrelated goals that are central to improving the understanding and management of organisational structure data. These objectives are methodically developed from the identified problems as well as the research question and lead to the implementation of a practical solution.

As a first objective we need to derive the required organisational structure data from the identified problem: This objective focuses on systematically extracting the required data elements related to organisational structure. It begins with a thorough analysis of the underlying problem to ensure that the derived data is both relevant and critical to solving the identified problems.

Second we need to identify source systems for organisational structure data: A key element of this thesis is the identification of different sources from which organisational structure data can be obtained. This involves a comprehensive investigation of potential data repositories and platforms and assessing their suitability and relevance to our research requirements.

To learn from the years of experience other companies might posses from developing the identified source systems containing organisational structure data, the third objective is to analyse the storage structure of these source systems.

The fourth objective is to design a uniform structure to standardise organisational structure data. This structure should provide a consistent framework for representing organisational structure data that allows for easier storage and consistent use for analytical purposes. Given the problems mentioned above, the unified structure should be designed in a way to support these three use cases:

1. Returning the immediate department and all higher-level departments given a specific contributor
2. Summarising all contributions of developers in the department and all subordinate departments for a given department
3. Summarising all contributions of developers in the department and all sub-

ordinate departments for every department given an organisation

After the uniform structure is created an approach shall be designed how this structure can be put into action in a way, that every system containing organisational structure data or a need for it can be connected to the framework. Given this flexibility, the elements should be standardised so as many code as possible can be reused for different source or downstream systems.

The final goal is to demonstrate the practical application of the research results through an exemplary implementation, especially in the context of Active Directory (AD). This will serve as a tangible proof-of-concept that illustrates how the standardised language and data abstraction methods can be used effectively in a real organisational structure.

In summary, this thesis attempts to bridge the gap between theoretical understanding and practical application in the area of organisational structure data management. By achieving these goals, it aims to contribute to the field by providing robust methods and tools for organisations to optimise their structural data analysis and management strategies.

4 Solution Design

This chapter presents the solution design, which is divided into two phases. In phase one, the first identified problem in form of the need for a uniform structure for organisational structure data will be addressed. The needed information on an organisation's structure is derived from the identified problem to solve it. Therefore a uniform structure will be designed and possible technical implementation option will be discussed. In phase two, a concept for an implementation of the proposed uniform structure is proposed. This concept uses the uniform structure from phase one as a critical component.

4.1 Phase 1: Designing a Uniform Structure for Organisational Structure Data

The goal of phase one is to design a solution addressing objective one, the need for a uniform structure to store and process organisational structure data. To achieve the highest quality solution possible, processing is carried out in four steps:

4.1.1 Derivation of the Required Organisational Structure Data

In this first step, we will identify which data is needed to gain the ability to map an individual contributor to every level of a company. For this task, we will assume the full name and the email address of the contributor are provided, as shown in the prior researchBuchner (2023). Based on these assumptions, the problem identified can be broken down into three parts:

1. The affiliation of a developer with his immediate department emphasises the closest organisational level to which they directly contribute.
2. The connections between departments on different levels, namely the affiliation of one department to another.

Let us consider point one first. For this connection, we need a data field in the user's data record containing his immediate department or a list of the associated employees for every department in the data set. Assuming most organisations follow some form of a tree-shaped organisational structure, we call a connection via an entry in the developer's data record the bottom-up approach, while the connection via the departments' employee list will be named the top-down approach. Which of these approaches works best for the uniform structure is subject to discussion in step three.

To gain the ability to map the connections between departments, we can similarly follow a top-down or bottom-up approach: Either the data record for the department contains some information about a head department or the information of subordinate departments to the one currently under examination is needed.

As a base of this mapping, some fundamental data on the employees and the departments as such is required. Only then can a known contributor - identified by his full name and email address described above - be mapped to the related employee in the database. To sum up the contributions on various organisational levels, we also need some information about the departments to connect here. We, therefore, need the following additional data:

- Master data of the employees.
- Master data of the departments.

For data protection reasons, this data can be filtered down to the employees and departments of interest before the data is extracted.

With these two sets of information, a complete connection of a developer to every level of the organisation is possible as long as one exists. Connections to other departments the contributor is not directly or indirectly working for are, of course, neither possible with the described information nor part of the objective and, therefore, ignored.

4.1.2 Identifying Possible Sources Containing the Required Data

The next step in designing a uniform data structure is to examine different systems typically possessed by a company for the required data. To simplify the scope of this thesis, we limited our search to evaluating three representatives of three different system types:

- SAP as the most commonly used ERP-System in German companies (19,3% market share)
- AD as the predominant entity in the domain of directory services respect-

ively the identity & access management process (>95% market share Germany)

- Salesforce as the most widely used Customer Relationship Management (CRM)-System in Germany (23% market share) (Statista Research Department, 2024b)

Of course, most companies possess many more IT systems storing parts of the relevant information or even have organisational structure data in the form of diagrams. However, since this would go beyond the scope of this thesis, these possibilities are not further evaluated. Our goal is to design a unified structure capable of representing a company's organisational structure through unified data. Therefore, we can assume that if one system of a kind, e.g. an ERP-System, contains the needed data, many other systems with similar functions or specifications can be used as well.

SAP

To start with SAP, some limitations need to be discussed since multiple SAP versions are used in the market. Every version comes with different modules and functionalities, so the research results may not be applicable to other versions. The currently most commonly used SAP version is SAP ECC, which stands for ERP Central Component; according to the DSAG-Innovation report of 2022, around 75% of all companies using SAP as their ERP-Systems are using SAP ECC. Around 4% of these are migrating to the newer SAP S/4HANA every year since SAP ECC will only be supported by the SAP company until 2027. Therefore, we will focus on SAP ECC for this research and give a short overview of SAP S/4HANA.

The relevant SAP module for this research is human resources and organisational management. For SAP ECC, this module is either SAP Human Capital Management (HCM) or SAP Human Resources (HR). The latter is the older version, which only contains the core features. Since all needed data is present in both in the same way, these differences are no concern to our research.

Our investigations for HCM show that the system is designed to contain all required data. The mapping of an employee to his immediate department is part of the sub-module Personal Administration (PA). There, the table PA0002 is used to store the personal information of every employee, including full name and email address. Table PA0001 then contains an employee's connection to his job key and organisational unit. The latter is the form of SAP storing information about departments. The connection is structured as a n:m (many-to-many) relationship table, the standard approach for managing many-to-many relationships between two tables in a relational database. This approach will be further investigated in step 3.

SAP uses the sub-module organisational management (OM) to store information about organisational units and connect one to another. Table HRP1000 contains all information about organisational units and objects of the type position, job, and cost centre. Table HRP1001 contains the connections between two objects of any type, including the connection between organisational units and organisational units and cost centres. Analogously to Table PA0001, the connection is structured as an n:m (many-to-many) relationship table between the objects stored in Table HRP1000.

As a result of this investigation, we need to extract the relevant parts of the four tables.

- Table PA0002 for personal information about the employees.
- Table PA0001 for the affiliation of an employee to his immediate department.
- Table HRP1000 for the master data of the departments.
- Table HRP1001 for the connection between different departments.

How this extraction can be achieved is part of Chapter 5 implementation.

SAP also uses employee groups, employee circles, and cost centres. These might be crucial for related problems, but our investigations show that these are not used for employee-department connections. Therefore, this data will not be used in this thesis.

Active Directory

The second is AD. Similar to SAP, AD is designed to contain all the needed information. In AD, employees are referred to as entities, while multiple options exist to map departments in the data. For entities, there are two major AD types: users and devices. Since our objective concerns the developer's contributions, we will only focus on users.

Since the specifics of AD's internal storage structure are hardly known to the public, we will refer to the data structure presented by the GRAPH API in this research. The GRAPH API is one of the primary ways to access and modify data in AD and is provided directly by Microsoft as a part of the AD system. Therefore, the data structure presented by this API is the best viewing angle for our further research in this thesis. Here, the user entity has a number of attributes assigned to them. These attributes contain the personal information of the employees they are presenting, including full names and one to many-email addresses. In addition, the field memberOf can contain the group IDs the respective employee is assigned to.

The object's organisational unit and group can be used for mapping departments and different hierarchy levels. While organisational units are the primary way to implement departments of all levels, the groups assign entities in a secondary structure. These can, for example, be the assignment of authorisations and access rights or the mapping of specific skills, e.g. in a matrix organisation as many consultancies are structured. While these groups might contain information about a company's hierarchy, we will primarily focus on organisational units due to this separation of functions. Like users, these organisational units contain several data fields like ID, a given name and most interestingly, a member and a members field. These are the two fields used to store the connections between two departments and a member's connections to a department. Users and other organisational units can be stored in the members list. We hence have the relevant data in only two places:

- The user table for the personal information connects a contributor to its employee data and connecting the employee to his immediate department.
- The table of organisational units responsible for storing the master data about departments, their sub-departments and assigned employees as well as the super-ordinate department.

Salesforce

Our last research is on Salesforce, a well-known CRM. Even though this kind of system is most likely only used by the sales-related departments of an organisation, Salesforce implemented steps to cover more and more of an organisation. The latest addition to this is work.com, a platform built to help companies overcome challenges related to the return to work during the COVID-19 pandemic. It might, therefore, be worth a look in case the other systems are not present in a company or the needed tables are not used.

Like SAP, Salesforce structures its data storage as a relational database. Due to the focus on customer and prospect data, the internal structures are quite different. Regarding employees, a table user is representing the employees of a company. It contains personal details as well as a field for a department. On the other hand, there is no separate table for departments. The functionality of the existing division table is designed to manage individual sales organisations with their accounts and leads. The only possibility to map departments to the system would be to use either a custom object or the accounts table intended for customer companies. Accordingly, it can be stated that a developer could be found in the system, mainly if work.com is used, and it might be possible to connect him to his immediate department. However, connecting to different hierarchy levels with the available data seems complicated. The conclusion, therefore, is that a CRM system is not the best place to search for organisational structure data. The

reasons are that the system is usually only used by the sales-related departments, and even though Salesforce has already extended this barrier, the needed data is still lacking. Hence, Salesforce as the market leader is insufficient, and it can be assumed that other CRM systems also fall short in this regard.

Summary

Finally, it must be said that introducing an SAP or Salesforce system is usually a major project. Large sums are often spent on specific adaptations of the systems to the company's respective processes. For this reason, it is not possible to say whether the table structure just described exists and is used in all companies that use these systems. Regarding AD, these adjustments are usually significantly less, but here, too, the data should be validated in advance for each case.

Therefore, Enterprise Resource Planning (ERP) and directory service systems should be the first choice when searching for organisational structure data in a company. Therefore, we will dive deeper into these two systems and perform a demonstration on AD in Chapter 6.

4.1.3 Analysing the Storage Structure of the Data

To start with this investigation, the first difference that seeks attention is the difference in the storage models. While SAP and Salesforce use a relational database, AD uses an object-oriented model. The consequence is that AD follows a hierarchical structure with a domain at the top and all organisational units, groups, and containers organised below. In contrast, SAP utilises tables that variably serve as the core of a snowflake-shaped data model. However, it has no singular central point to which all other objects are subordinate. Instead, the connection tables PA0001 and HRP1001 explicitly serve as n:m (many-to-many) relationship tables. This means SAP is designed to reject a tree-shaped structure of the organisation regarding employee-department connections and department-department ones. AD follows this example on the level of memberships by structuring the data fields "memberOf" and "members" as lists with potentially more than one entry. Thereby a many-to-many relationship can be mapped as well. It is, therefore, fair to say that both systems followed the need to represent more complex organisational structures, deviating from a purely tree-like structure at one point. Because of that, our uniform structure also needs to support these many-to-many relationships.

The second point is that both storage structures allow for easy traversal of the organisational structure in both directions: Starting from a specific user upwards to all departments to which he is a direct or indirect member, as well as downwards from a given department to all sub-departments and subordinate employees. This feature is also essential when considering the three use cases the uniform struc-

ture is supposed to support according to Chapter 3: If we have an organisation or a department given and want to sum up all contributions, a traversal downwards is necessary. On the other hand, if a user is given and his memberships are to be discovered, an upward traversal through the organisation is required.

So in conclusion the following two findings can be summarised:

- Both data structures are able to map many-to-many relationships regarding department-to-department as well as employee-to-department connections.
- Both data structures allow for easy traversal in both directions, upwards and downwards.

4.1.4 Designing a Uniform Structure

In the final step of phase one, the knowledge gained is used to design a uniform structure. We will first summarise the most critical cornerstones of requirements for the uniform structure discovered in the last chapters and sections to make the most advantageous decisions possible when designing the uniform structure. From these, we can then derive essential criteria for the uniform structure. We will look at different implementation possibilities with the criteria and choose the most optimal combination. As a result, we will get the ideal uniform structure for organisational structure data.

Cornerstones of Requirements

As the object definition describes, the goal is to design a uniform structure as a standard. All organisational structure data from different sources must be translated into a uniform structure. The uniformed data can then be used in any other system. At the same time, this downstream system needs no further adaptation or knowledge about the system from which the organisational structure data originates.

Second, the identified problems left us with three use cases the uniform structure needs to support. As discussed at the end of Chapter 3, these can be summarised as getting the sum of contributions for various organisational levels and receiving all immediate or super-ordinate departments for a developer.

We also learned from the analysis of the storage structure in the source systems that these systems are designed for various complex organisation structures, and, therefore, these should be accounted for.

As a fourth point, the structure needs to be viable with large data sets of a couple hundred or even a thousand employees since many companies will have this size. Regarding software development companies, most of these employees may be relevant to the identified problems.

Important Criteria for Design Choices

As a result, the following criteria can be derived:

1. All data identified as necessary in section 4.1.1 must be included.
2. The uniform structure must allow for (efficient) upward and downward traversal.
3. Many-to-many relationships between all entities must be able to be mapped in a uniform structure.
4. Memory utilisation and access times must remain low even with complex structures, ideally independent of the complexity.

The first three criteria must be met for the uniform structure to fulfil the objectives. The fourth criterion is more continuous since it will not be possible to map all three use cases in constant time. Therefore, memory requirement and runtime will be evaluated with the Big O Notation, a mathematical concept used to classify the complexity of algorithms by giving an upper estimate depending on the input complexity. The three use cases are the relevant measuring points for evaluating memory requirement and runtime.

- The query of all departments a given user is a member of, with d being the number of such departments.
- The sum of all contributions for a department made by developers of this department.
- The sum for all departments in an organisation, with d being the number of departments.

Evaluation of Design Possibilities

As identified in 4.1.1, the two main components of the uniform structure will, on the one hand, be the master data for employees and departments. On the other hand, the connections between these entities needs to be mapped. For both categories, different implementations might be best.

To begin with the master data, it becomes obvious that organising this data as a dictionary with key-value pairs is the best choice. While the identifier for the master data record (e.g. the employee's email address) is used as the key, all other entries can be stored as a sorted list and inserted as the value. These values can then be organised as a dictionary once again to allow direct access to the data stored for every entity. There would be a separate dictionary for each type of entity to consider the differences between users and departments.

The reason for the superiority of this choice is that dictionaries can be implemented using hash tables. These allow for an average case search, insert and delete time of $O(1)$ with $O(n)$ storage space needed, given n as the number of master data entries, e.g. employees or departments. The next best option would be a balanced binary search tree like an AVL-Tree, an average case search, insert and delete time of $O(logn)$. The advantages of such trees are the deterministic performance and storing data in sorted order. Deterministic performance refers to the fact that these trees keep the time complexity at $O(logn)$ in the best and worst case, while the worst case time for hash tables is $O(n)$ due to possible hash collisions. However, since the risk of this worst case can be mitigated using suitable hash functions and dynamic rescaling of the hash table, and the data will not be needed in sorted order, a dictionary seems the best choice.

One issue with both concepts is our stipulation that employees should be identifiable by their full name and email address. Since a hash table only uses one key and the search in all values needs $O(n)$ time, this poses a problem for the efficiency of the implementation. There are two reasonable solutions to this issue: The first is to use the email address as a key and store how an email address is created for an employee. Since most companies use the full name of the employee to create the mail address, e.g. philipp.winklmann@companyname.com or pwinklmann@companyname.com, this should cover most cases while the rest of the cases can be searched with $O(n)$ in the values. The second option would be to create two hash tables, one with the email address and one with the full name. Both lookups will then point to the same stored value data for this employee. That way, we can use both identifiers as keys and only need to store the keys twice, not the values.

For mapping the connections between these entities, two different options emerge. The connections could either be included in the value parts of the master data dictionary on both sides of the connection, or additional many-to-many connection tables could be added to map the connections externally. While AD follows the first approach, SAP sticks with the second option since generations of SAP systems due to the usage of relational databases. To evaluate these two options for our use case, we will compare the runtime and storage space needed for both options and our three use cases. Option one is to be named embedded mapping; option two is called mapping by a separate association table.

For option one, we would have to add one entry to the user's value dictionary, a list of departments to which the user belongs. Three more entries would be necessary on the department side: one list of employees belonging to this department and two lists for the super-ordinate and the subordinate departments. Doing so will implicitly turn all the single entities into graph elements that know their neighbours.

As well researched in graph theory, the lists can be turned into sets to improve

performance further. Sets are represented internally as hash maps or balanced trees and therefore make it possible to query in $O(1)$ in the best and average case or $O(logE)$ in the case whether a given entity, e.g. department or employee, is connected to another entity. E is, thereby, the number of connections. The implementation as a set has no disadvantages for our use case since the time to visit all connections is still $O(E)$. [1]An example for option one can be seen in table 4.1 representing the organisational structure seen in figure 4.1.

Entity	**Entries in entity's master dictionary**
Employee 1	Departments = Set(3)
Employee 2	Departments = Set(1)
Employee 3	Departments = Set(1, 2)
Department 1	• Employees = Set(2, 3) • Subordinate departments = Set(2, 3) • Super-ordinate departments = Set()
Department 2	• Employees = Set(3) • Subordinate departments = Set() • Super-ordinate departments = Set(1, 3)
Department 3	• Employees = Set(1) • Subordinate departments = Set(2) • Super-ordinate departments = Set(1)

Table 4.1: Example of an adjacency set to be included in entity's master data dictionary

These four sets or lists can then contain only the direct connections or all indirect connections. The latter would mean we store the immediate department for a user and all super-ordinate departments to his immediate department. By doing so, the runtime for the different use cases can be reduced by the cost of requiring additional storage for storing each department and its connections multiple times. The same logic can be applied to the connections between departments. This procedure would imply redundant data storing- a highly unpopular practice due to the risk of data inconsistency. In addition, the time complexity for adding or editing data and the required storage would increase. The benefit would be a reduction of time complexity for lookup operations from $O(E)$ to $O(1)$, i.e. from linear to constant time.

Looking at the three identified use cases, this choice would mean a reduction in time complexity for

[1]Please note that there is a significant distinction between visiting all connections and returning all connections. While visiting all connections has the complexity $O(E)$, returning the set that contains all connections can be done in $O(1)$ as described in the use case complexities.

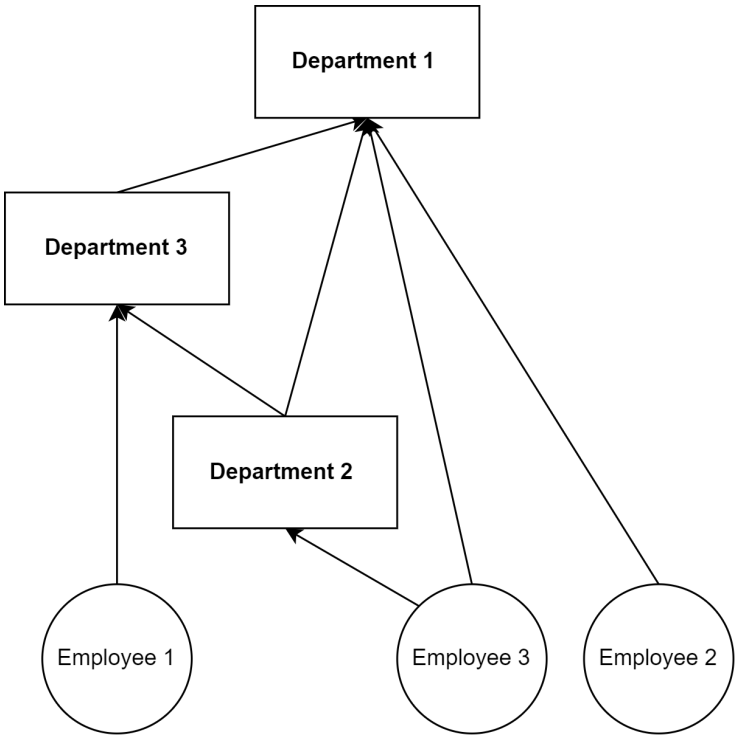

Figure 4.1: Examplary complex organisation structure

- use case one from $O(d)$ to $O(1)$ with d being the number of departments in the organisation,
- use case two from $O(m+d)$ to $O(m)$ with m being the number of employees in the organisation, assuming the contributions are already summed up per employee. Since $O(m + d) = O(m)$, assuming most companies have way more employees than departments, there is no improvement.
- use case three, there is no reduction in time complexity. The time complexity in both cases is $O(m + d)$.

Let us briefly examine how these runtime complexities come about. For the use case, one option, 'only immediate connections stored', the immediate department is in the dictionary of the employee master data, and the complexity is, therefore, $O(1)$. We can get the following super-ordinate department from the immediate department in $O(1)$ and must traverse up a maximum of d times if d is the number of departments in the company. Therefore, the complexity is at most $O(d)$. For option two, 'all connections stored redundancy,' we need to look up the list of departments in the employee master data dictionary, which is possible in $O(1)$. For the use case, two options, 'only immediate connections stored,' the fastest way to sum up all contributions is to make use of the fact that every employee is only to be counted once, even if he is associated with multiple departments.

We can hence traverse down from the department a maximum of d times and get the associated employees for every department. Adding them to a unique set has an average complexity of $O(1)$ since sets can be implemented as hash tables, similar to dictionaries. Once the unique set of employees is formed, the look-up of every employee's contributions will take at most m times $O(1)$ time, with m being the number of employees at the company. The complexity is, therefore, $O(m+d)$. Suppose all direct and indirect associated employees are stored in every department's value dictionary. In that case, the time is $O(m)$ since only the look-ups for at most n employees are necessary. The time complexity for use case three is, in both cases, $O(m+d)$. That is the case because the fastest way is to add the contributions of the immediate employees for every department in $O(m)$ and then summarise each department's sum from the bottom up to calculate the sum of the following higher department. This implies another d many summations, which leads to $O(m + d)$. This way is even faster than calculating each department's sum in $O(m)$ for option two since this would lead to a total of $O(m*d)$. To enable this bottom-up approach, we need to keep track of all departments with no superordinate department above them to start the algorithm from there, perform a full traverse down and sum up all contributions on the way back up.

In conclusion, we can say that by storing the connections redundantly, there is only a significant improvement in use case one. In addition, the number of departments in an organisation can be expected to be significantly lower than the number of employees. The improvement shown is therefore not considered necessary or relevant enough to justify the data redundancy, the additional storage needed, or the additional complexity for adding or editing data. Accordingly, we will use the lists with direct connections only for further comparison.

Next, we will consider option two, 'mapping by separate association table'. For this, different implementations are also conceivable. Since our problem of mapping the connections between entities is very similar to mapping all edges between nodes in a graph, we will examine which of the well-researched solutions in graph theory fits our problem.

The first exciting solution in graph theory is an adjacency list or set. An adjacency list consists of a collection of lists, where each list contains the adjacent vertices for one node. This structure is efficient for sparse graphs, as it stores edges directly connected to each node, allowing for quick access to a node's neighbours. In an adjacency set, these lists are replaced by sets, allowing for faster checks on whether a particular node is among the adjacent vertices of another node. It becomes pronounced that this structure is equal to the option discussed before, adding the connections directly to an entity's master data. Adding an adjacency list would not deliver any benefits and only require additional storage and slow down the access of an entity's connections.

The second well-known solution is the adjacency matrix. An adjacency matrix

is a square matrix used to represent a finite graph. The matrix's element (i, j) indicates whether node i and node j are adjacent or not in the graph. The adjacency matrix is an VxV matrix for a graph with V nodes. Our specific case with connections between super-ordinate and subordinate departments or employees that belong to a department is similar to a directed graph. Here, graph theory shows that the best option is to implement the adjacency matrix as a binary matrix where $(i, j) = 0$ if nodes i and j are not connected and $(i, j) = 1$ only if j is the super-ordinate department of department i respectively j the immediate department of employee i. Suppose the edge is directed in the opposite direction, only $(j, i) = 1$ while $(i, j) = 0$. An example of such a table can be seen in table 4.2. This example would represent the structure seen in figure 4.1.

	Department 1	**Department 2**	**Department 3**
Employee 1	0	0	1
Employee 2	1	0	0
Employee 3	1	1	0
Department 1	-	0	0
Department 2	1	-	1
Department 3	1	0	-

Table 4.2: Example of an adjacency matrix representing all connections in one table

One core principle of an adjacency matrix is the lookup of one or all connections for a given node in $O(1)$, which is usually possible by knowing the order in which nodes are represented in the matrix. We need to find a workaround to achieve this for our use case since our data is not naturally contiguous. We can add an index to the master data for each entity or create one dynamically using a hash function.

For the particularity of our case, where we have two different entities, an employee and a department, we can make some changes to the basic concept of an adjacency matrix. It is unreasonable to create two separate adjacency matrices for the two connection types because departments play a crucial role in both. The storage would, therefore, only inflate while the access times would still be $O(1)$. The better option is to remove the employees from the columns and only add rows for them since we are neither interested in employee-to-employee relationships nor can an employee be super-ordinate to a department. The storage space for the adjacency matrix will therefore be reduced from $O((m+n)^2)$ to $O((m+d)*d) = O(m*d+d^2)$ with d being the number of departments while m is the number of employees of the organisation. This reduction is quite significant since the number of employees is usually much larger than the number of departments

in an organisation. Given this implication, the complexity can be reduced to $O(m * d)$ by the rules of the Big O Notation.

Two other useful tools to represent a graph known in graph theory are an incidence matrix and an edge list. An incidence matrix is a VxE matrix with V being all vertices and E being all graph edges. This matrix contains the information for all edges whose vertices are adjacent to this edge, thereby creating an edge-focused graph representation. This is useful for displaying hyper-graphs or analysing edge-node relationships. This representation is not reasonable for our use case since the access time for all connections of one entity is complex due to the nature of the representation. The storage space having a complexity of $O(VxE)$ is also similar to the complexity of an adjacency matrix, assuming every employee and most departments have at least one super-ordinate department. An edge list is a collection of all the edges in a graph, where each edge is represented as a tuple (or a triplet for the direction in our case) of adjacent nodes. This could be implemented as a list of triplets for our case at hand, leaving us with a time of $O(E)$ to check whether a given entity is super- or subordinate to another given entity. We would organise the list of triplets as a balanced tree or a hash map to improve this access time. This would improve the average access time to O(1). The only downside is the access for all connections to a given entity: Due to the storing of a connection as separate triplets for every connection, we first need to find the right place in the tree or hash map ($O(E)$) to then access all connections ($O(n)$).

To summarise all this research, we can state that various technical approaches have different strengths. An overall comparison of all considered implementation options can be seen in table 4.3. Since it is reasonable to assume that an entity in an organisation has no more than two super-ordinate departments on average, our number of edges is more minor than $(n * logn)$, a threshold for a dense graph in the literature. Therefore, the adjacency matrix cannot fully play its strengths and is rejected due to the high storage and slower access time for all connections of an entity. On the other hand, the integrated adjacency list benefits from this relative sparsity of the graph and impresses with low access times and storage usage. The edge hash map has a clear lead for the external options with a separate connection table due to its low complexity in access times and storage.

Conclusion: The Ideal Uniform Structure

As discussed in the last section, implementation has many different possibilities. Each possibility has its strength in different scenarios. We now want to evaluate the earlier set criteria and choose the best approach for our uniform structure.

Regarding our criteria the results show that

1. All data identified as necessary in chapter 4.1.1 can be included is all options

Implementation	Test specific direct connection	Visit all connections	Storage space
Set with immediate connections in master data	$O(1)$	$O(E)$	$O(E)$
Set with all connections in master data	$O(1)$	$O(E)$	$O(E^2)$
Adjacency matrix	$O(1)$	$O(E^2)$	$O(n^2)$
Incidence matrix	$O(E * n)$	$O(E * n)$	$O(E * n)$
Edge hash map	$O(1)$	$O(E + n)$	$O(E)$

Table 4.3: Summary of all considered implementations for average case

presented.

2. All options presented allow for (efficient) upward as well as downward traversal.
3. Many-to-many relationships between all entities can be mapped in all options presented.
4. The best options regarding memory utilisation and access times can be seen in 4.3.

Therefore the decision comes down to criteria four. Given the above made considerations and assumptions, the most suitable structure emerges distinctly. For storing the master data for employees and departments, we use dictionaries and include additional sets to store the connections inside these dictionaries. Therefore, the final structure for employees and departments can be seen in figure 4.2. The complexity for this data structure is as shown in table 4.4.

Use case	Complexity of proposed structure
Get all departments for a given user	$O(1)$
Summarise all contributions for one given department	$O(m + d)$
Summarise all contributions for every department	$O(m + d)$
Storage space complexity	$O(E)$

Table 4.4: Summary of runtime and storage space complexity for predefined use cases

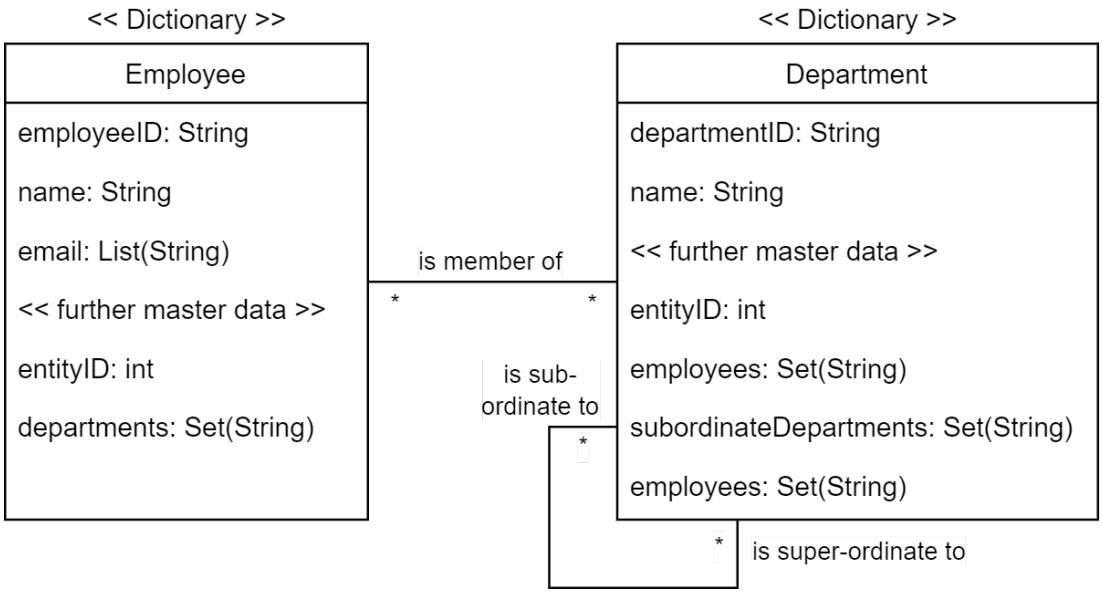

Figure 4.2: Overview of the final data structure

4.2 Phase 2: Bringing the Concept to Life - Proposing a Concept to Deploy the Uniform Structure

A concept is proposed to bring the theoretical concept into action regarding how a program should be structured to fulfil the defined objects. The implementation supplied with this thesis will be implemented in the programming language Python due to multiple advantages. These include among others

- **Easy data connection:** Python offers strong support for various data sources and formats, including SQL and NoSQL databases, CSV and Excel files, and APIs.
- **Flexibility and scalability:** Python is suitable for small data sets and large, complex application systems.
- **Rich libraries:** Python has a large ecosystem of libraries and frameworks ideal for data manipulation, such as Pandas, SQLAlchemy or NumPy, to name the basics.

The implementation can also be implemented in any other programming language using the following description.

The Structure of the Implementation

The programm will be structured into four main modules:

- **Requestor** responsible for extracting the data from the data source, i.e. the source system of the organisational structure data.
- **Unifier** to transform the data from the data structure of the source system to the unified structure proposed in this thesis.
- **Transformer** transforming the uniform data to the required data structure for the write operation, e.g. a relational data structure to write the organisational structure data to a SQL database.
- **Writer** responsible for writing the data to the intended storage location.

This modular system allows users to create new modules for different applications independently. This can be source systems as well as downstream systems. Therefore, including new organisational structure data from a new data source only requires a new Requestor and, most likely, a new Unifier. At the same time, the existing Transformer and Writer modules can be reused. In addition, not all modules need to be used. If the data is supposed to be stored in a unified form, no transformer will be needed. This allows users to extend the implementation with modules suited for their application.

From these modules, any number can be assembled into pipelines. These pipelines also define how the data is forwarded from one module to the next inside the pipeline and allow for fast access to two regularly used modules. Controlling the program sequence is the orchestrator. It contains the primary method, and here, all modules and pipelines can be chained together to achieve the desired result.

The implementation supplied with this thesis will contain a Requestor and Unifier for AD as described in Chapter 6, as well as essential Transformers and Writers to store the data to a JSON file in the unified structure or the original data structure of the source system.

The specifics of how the implementation is done will be explained in Chapter 5.

5 Implementation

The goal of this chapter is to focus on the practical implementation of the solution developed in Chapter 4 "Solution Design". For this specific case, this means creating a detailed implementation for the abstract solution designed in section 4.2. This is intended to enable the practical use of the uniform structure proposed in section 4.2. As described abstractly in section 4.2 the main elements of the implementation are

- **modules**: The four types of modules Requestor, Unifier, Transformer and Writer
- **pipelines**: To connect multiple modules that are frequently used together into sequential bundles
- **orchestrators**: The central control units where the modules and pipelines are chained together to achieve the desired outcome.

5.1 Overview of the Implementation in Detail

Before we give an overview of the implementation, it must be stated that the structure described in this section was not created for this thesis from scratch. The MecoIS team provided the basic structure, a project currently in development at the Professorship for Open-Source Software at the Friedrich-Alexander University Erlangen-Nürnberg.

To go into further detail on the implementation, we will start with the overall folder and package structure. On the highest level are the folders data to store data, e.g. in flat files and pipelines containing the infrastructure. Inside the pipeline folder, the following structure emerges:

- **controller**: A support structure explained further down this chapter.
- **doc**: For the documentation of the implementation.
- **modules**: Containing all types of modules.

  - **requestor**
  - **unifier**
  - **transformer**
  - **writer**
- **orchestrators**: Containing all orchestrators.
- **pipelines**: For all pipelines.
- **schema_files**: As explained later, mapping different data structures.

Let us now have a closer look at all the elements starting going from top to bottom in the folder structure: In the controller folder are the implementations of the Pipeline class and the PipelineStep class. These two classes provide a framework for all pipeline implementations in the pipelines folder. The Pipeline class, named PipelineSteps, is designed to manage and execute workflow steps in sequential order. It provides mechanisms to coordinate the execution flow and pass arguments between steps. Key functions include

- Initialisation with options for execution intervals, pipeline name, and loop mode.
- Dynamic addition of pipeline steps.
- Execution control that supports immediate and interval-based looping execution modes.

The PipelineStep class represents single steps within a pipeline, e.g., any named modules. A step is defined by an object and method to execute. Initial arguments can be passed along and combined with forwarded arguments from the last pipeline step during execution.

The doc folder only contains the documentation for further insights into the implementation. The documentation is further discussed in section 5.2.

In the orchestraters folder, any number of orchestraters can be stored. Orchestraters are the central control units where all the threads come together. These files work as a pipeline by creating an object of the Pipeline class and filling it with different steps - pipeline implementations from the pipelines folder or any module type. Therefore, an orchestrator imports the classes Pipeline and PipelineStep and all the chosen modules and pipelines. After preparing the pipeline, it is executed to perform all types of tasks.

Next up are modules. These are the most minor components in the framework, each performing a single task. They divide the process from extracting the organisational structure data to storing it in a new structure into four parts, as described. Each type has an abstract implementation to ensure the interconnectiv-

ity of different modules and standardise the interfaces and methods required. The abstract implementation of every module type contains its type-specific method, e.g. write for Writer, unify for Unifier, etc. Each of these methods gets three arguments passed along from the pipeline. The first is the PipelineStep that is currently executed. This enables the implementation to call pipeline_step.next() on the given argument pipeline_step to execute the next step in a pipeline. Second, the result of the last module is passed down by the pipeline. This data was extracted or modified in the last module and is now to be further processed. The third argument is the initial transfer when the pipeline is created, e.g. in the orchestrator. Table 5.1 gives an overview of the four module types. A speciality is the Requestor type since it usually starts a pipeline and has no upstream modules executed before. Therefore, no forward argument is passed along. In addition, a requestor has four specific methods a developer can choose from when creating a requestor. These are tailored to the specifics of different request types and can be used equivalently.

Module type	**Specific methods**	**Forwarded argument**	**Initial arguments**
Requestor	• sync_request() • delta_request() • async_request() • schema_request()	-	custom_args
Unifier	unify()	source_data	source_schema
Transformer	transform()	uniform_data	target_data
Writer	write()	content	details

Table 5.1: Overview of the module types and their specific methods

In the pipelines folder, all pipelines that developers can create from multiple modules are managed. A pipeline implementation in this folder serves as an additional level of abstraction between modules and orchestrators and is very similar to an orchestrator implementation. There is an abstract_implementation class for pipelines to ensure that all vital functions are present to enable the interchange between different modules and pipelines, there is an abstract_implementation class for pipelines. The abstract setup_pipeline method is defined, which every pipeline must implement. Additionally, the execute_pipeline method is implemented and ensures the proper execution of the pipeline once an orchestrator pipeline commands it to do so. A pipeline in an orchestrator can simply be executed using this abstraction level. It will make sure to execute either the specific function in a module or a subordinate pipeline for every element in the pipeline, no matter the type. Therefore, every module only needs to call the following method on the given argument pipeline_step, and it will be executed whether the next element is a module or a pipeline.

Last, schema files can map data types and names to their respective counterparts in different data structures. These schema files can be used by either a Unifier to transform the data structure from the structure of the source system to the uniform structure or by a Transformer to map data in the uniform structure to the desired output structure. The files are organised as YAML files divided into two parts, "NameTypeMapping" and "NameMapping". One file for each mapping and direction is created and stored in the schema_files folder.

5.2 Documentation

The documentation of the described implementation consists of three parts: Every file contains documentation comments. These are special comments that can be used to generate documentation automatically. They describe the file's purpose and use of every class and method.

Second, architecture documentation is provided in the doc folder. This provides a more detailed overview of the described architecture, including the technologies used, system structure, components, and their interactions.

Third, a basic user guide is provided in the form of a README in the doc folder to explain the implementation's further usage and expansion possibilities.

In addition, inline comments are used wherever further explanation of specific steps is necessary.

5.3 Overview of the Implementation Elements Provided with this Thesis

For the demonstration of the concepts proposed in this thesis and to give future users an implementation with all needed basic functionalities, an implementation is provided with this thesis. Here, we will give an overview of which functionalities are already implemented and ready to use.

First, the controller for Pipeline and PipelineStep are implemented so pipelines can be created. Furthermore, the abstract implementations for pipelines and all four types of modules are provided and can be imported for further creation. Regarding operative implementations, there are three Requestors to import data from an AD instance, as well as read data from a JSON file in the source systems data format (i.e. stored after any source system requestor before processed by the corresponding Unifier) or in the uniform structure. For Unifiers, there is only the Unifier from the AD data structure to the uniform structure as needed for the demonstration in Chapter 6. No Transformers have yet been implemented. There is one writer to write the given data to a JSON file. For this writer, it

is of no point whether the data is in a uniform structure or the source system structure.

These modules are chained into two pipelines. The first pipeline, "AD_extract_pipeline", imports data from AD and stores the raw data in a JSON file. The pipeline "AD_unify_pipeline" then reads this data from the JSON file, unifies it to the uniform structure and stores the data to another JSON file.

All these steps are put together in an orchestrator "organisation_orchestrator" where the pipelines are assembled into the main pipeline, and the necessary connection details for AD and the storage locations for the JSON files are provided as initial arguments.

Many of the provided elements are described in more detail in Chapter 6. Table 5.2 shows an overview of the provided elements.

Element type	**Elements provided**
Controller	• Pipeline • PipelineStep
Module: Requestor	• abstract_implementation • AD_Requestor • raw_JSON_reader • enriched_JSON_reader
Module: Unifier	• abstract_implementation • AD_Unifier
Module: Transformer	abstract_implementation
Module: Writer	• abstract_implementation • JSON_writer
Pipelines	• abstract_implementation • AD_extract_pipeline • AD_unify_pipeline
Orchestrators	organisation_orchestrator

Table 5.2: Overview of all elements provided with this thesis

5.4 How to Connect to Active Directory and SAP

In order to enable reads to connect to the two systems discussed in detail in this thesis and in preparation for the demonstration in Chapter 6, we given an overview of how to connect to AD or SAP and extract the organisational structure data needed for the identified problems.

Connection to Active Directory

First, we discuss how an AD system can be accessed using the GRAPH API to retrieve the relevant organisational structure data. This process requires several steps to extract the data successfully. For short, these steps are:

1. **Identify your AD Instance**: Begin by determining your organization's Azure AD instance, using the unique tenant ID.
2. **Register your Application**: Register an application within Azure AD to obtain a client ID and client secret, which are necessary for authentication.
3. **Authenticate and Authorize**: Utilize OAuth 2.0 to authenticate your application with Azure AD, acquiring an access token by presenting your client ID, tenant ID, and client secret.
4. **Request Data**: Use this access token for every request to the Graph API as an authentication. Now, you can access specific endpoints, usually starting with 'https://graph.microsoft.com/v1.0/' followed by the specific data you want to request. This could be /users for user data and /groups for departmental information. These are the two main datasets required for the identified problems above.
5. **Extract and Utilize Data**: Pass the returned JSON data to the corresponding Unifier to transform the data into a uniform structure.

The appendix has a complete guide on implementing these steps using Python.

Connection to SAP

Regarding SAP systems, data extraction is slightly more complicated. To achieve a similar live connection to the system, several complicated steps are required that need to be set up by the administrators of the system:

1. An SAP User needs reading rights to the necessary tables. This user will later provide access to the data.
2. A RFC module must be created and installed on the SAP system to prompt the user to extract and store the data on a network drive. This extraction will be polled by the SAP OPC and performed in chunks. The RFC module then pseudonyms the data if required and provides it to a client in the system.
3. In the company's local system, an on-premise client needs to be set up, e.g. on a virtual machine to check for extracted data and provide these to an external system, i.e. the developer's setup. This client is usually also used to translate the developer's requests to RFC.

Since these three elements are complicated to develop, data is usually extracted as flat files for easier projects, e.g., CSV or Parquet. This can be achieved in two ways. The first one uses a custom-made ABAP report where all needed tables, columns, filters and pseudonymisations can be configured. ABAP is the programming language for SAP and is needed to create such a report. Alternatively, there are creators on the internet that can be used by uploading an Excel file with the specifications. This method is preferred for large amounts of data. The even easier way is to ask a user with access to the SAP system to export the files manually using SAP SE16N transaction. Here, all parameters can be configured as well, and the results can be downloaded as Excel or CSV files. Since this method only allows one extraction at a time, it is a more time-consuming way for larger data sets.

6 Demonstration

This chapter aims to demonstrate the proposed solutions using organisational structure data from an Active Directory demonstration system as an example. At this moment, we will demonstrate both the uniform structure designed in section 4.1 as well as the implementation concept designed in section 4.2 and discussed in detail in Chapter 5.

The proposed architecture from section 4.2 will be used to build a Requestor and a Unifier for AD. The Requestor will use the GRAPH API provided by Microsoft to access the data from the AD demonstration system and pull the in section 4.1.1 identified data into the system. The Unifier will transform the data into the uniform structure proposed in section 4.1.4. In addition, a Writer will be implemented to store the uniform data to a JSON file.

6.1 Extraktion of the Data from Active Directory

The complete guide on how to extract data from an Active Directory System using Python can be found in Appendix 1, but here are the basic steps for overview:

1. **Identify your AD Instance**: Begin by determining your organization's Azure AD instance, using the unique tenant ID.
2. **Register your Application**: Register an application within Azure AD to obtain a client ID and client secret, which are necessary for authentication.
3. **Authenticate and Authorize**: Utilize OAuth 2.0 to authenticate your application with Azure AD, acquiring an access token by presenting your client ID, tenant ID, and client secret.
4. **Request Data**: Use this access token for every request to the Graph API as an authentication. Now, you can access specific endpoints, usually starting with 'https://graph.microsoft.com/v1.0/' followed by the specific data you want to request. This could be /users for user data and /groups for departmental information. These are the two main datasets required for the identified problems above.

5. **Extract and Utilize Data**: Pass the returned JSON data to the corresponding Unifier to transform the data into a uniform structure.

Once these steps are complete and the requests for users and groups was successful, the following data were output from the AD demonstration system:

Example for one user:


```
{
    "businessPhones": [],
    "displayName": "Conf Room Adams",
    "givenName": null,
    "jobTitle": null,
    "mail": "Adams@M365x214355.onmicrosoft.com",
    "mobilePhone": null,
    "officeLocation": null,
    "preferredLanguage": null,
    "surname": null,
    "userPrincipalName": "Adams@M365x214355.onmicrosoft.com",
    "id": "6e7b768e-07e2-4810-8459-485f84f8f204"
}
```


Example for one organisational unit:


```
{
    "id": "06f62f70-9827-4e6e-93ef-8e0f2d9b7b23",
    "deletedDateTime": null,
    "classification": null,
    "createdDateTime": "2017-07-31T17:38:15Z",
    "creationOptions": [],
    "description": "Video Production",
    "displayName": "Video Production",
    "expirationDateTime": null,
    "groupTypes": [
        "Unified"
    ],
    "isAssignableToRole": null,
    "mail": "VideoProduction@M365x214355.onmicrosoft.com",
    "mailEnabled": true,
    "mailNickname": "VideoProduction",
    "membershipRule": null,
    "membershipRuleProcessingState": null,
    "onPremisesDomainName": null,
    "onPremisesLastSyncDateTime": null,
```

```
    "onPremisesNetBiosName": null,
    "onPremisesSamAccountName": null,
    "onPremisesSecurityIdentifier": null,
    "onPremisesSyncEnabled": null,
    "preferredDataLocation": null,
    "preferredLanguage": null,
    "proxyAddresses": [
        "SMTP:VideoProduction@M365x214355.onmicrosoft.com",
        "SPO:SPO_16219fd2-fafd-4fea-8084-8b5eaa8c5ad2@SPO_dcd219dd-bc68-4b9b-bf0b-4a33
    ],
    "renewedDateTime": "2017-07-31T17:38:15Z",
    "resourceBehaviorOptions": [],
    "resourceProvisioningOptions": [],
    "securityEnabled": true,
    "securityIdentifier": "S-1-12-1-116797296-1315870759-261025683-595303213",
    "theme": null,
    "visibility": "Public",
    "onPremisesProvisioningErrors": [],
    "serviceProvisioningErrors": []
}
```


Further examples can be found in Appendix 2.

6.2 Unification of the Extracted Data

In the second part of the demonstration the above extracted data is to be unified. For this purpose two master data dictionaries with structures according to the uniform structure that was designed in section 4.1.4 are created. Now for employees and departments two different methods are started each in sequential order. The first method maps all data fields for the extracted users and departments to their new names and data types using the "AD_to_unified_schema_mapping.yml" schema file. If the lookup in this file doesn't result in a match the data field is either copied as is or not copied at all - dependent on the modules configuration.

Afterwards the connections are transformed into the proposed target structure. Since AD already uses a quite similar solution, this is a fairly simple task.

After applying these two steps the data now exists in uniform structure and can be stored or transformed to a target structure while the original export can be deleted.

7 Evaluation

In the concluding phase of the design science research methodology, the crucial task is to assess if the created artefacts, in this thesis, the uniform structure and the implementation approach, effectively address the initially identified problem, as outlined by Peffers et al. (2007). Therefore, this chapter first describes the evaluation methods used in section 7.1. Section 7.2 presents this evaluation's results, while the following sections will discuss the practical implications of this research, its applicability and expandability, and its limitations.

7.1 Evaluation Methodology

We employ a qualitative approach to evaluate the effectiveness of the concepts proposed in this thesis, namely the uniform structure and its implementation approach. Our primary goal was to create a uniform structure for organisational structure data to close the gap between known contributions on individual levels and the need to aggregate these on various levels of the organisation. We therefore evaluate the following aspects:

- Fulfilment of the objectives set in Chapter 2 "Object definition" and especially of the derived use cases.
- Compliance with the criteria established in the course of the research.
- Compatibility test with the data from suitable systems.
- Integration test with existing tools.
- Evaluation of whether the identified problems can be solved with the proposed solution

7.2 Evaluation Results

This section presents and discusses the evaluation results, subdivided into the different aspects described in the methodology section above. The evaluation

includes results for both proposed parts, the uniform structure, and the implementation approach. The evaluation yielded the following results:

7.2.1 Fulfilment of the Set Objectives

The first objective defined in Chapter 2 two is deriving the required organisational structure data from the identified problem. This objective was achieved in section 4.1.1 with the results being that four sets of data are needed to achieve further objectives. These four sets are the department-to-department connections, the employee-to-department connections, and the master data records for employees and departments.

Second, reliable sources for organisational structure data were to be identified. This objective was addressed in section 4.1.2. Even though two suitable sources were identified in the form of AD and SAP, only three possible sources were evaluated. This, for sure, is a point where further development would be appropriate. On the other hand, our results show that some types of frequently used systems in an organisation contain the necessary data, and derived from the identified problems, it is most likely that a company knows where to find organisational data once the problem occurs.

As a third objective, a unified structure was to be designed so that it would support the three derived use cases. These use cases were "Returning the immediate department and all higher-level departments given a contributor", "Summarising all contributions of developers in the department and all subordinate departments for a given department", and "Summarising all contributions of developers in the department and all subordinate departments for every department given an organisation". For this objective, a uniform structure was designed in Chapter 4 that supports all three use cases. For the first use case, the immediate departments can be obtained in the employee's master data record, and all higher-level departments can be identified by traversing the tree-like structure upwards. This can be done using the list of super-ordinate departments in every department's master data record. Use cases two and three can be achieved by a similar strategy in reverse order. Therefore, the start is either the specified department (use case two) or all departments with no super-ordinate department stored in a list. From there, all immediate employees will be extracted using the employee set in the department's master data record, while the subordinate department's list is used to traverse the structure downwards and repeat the process with all subordinate departments. Once a list of employees in the substructure under consideration is complete, all contributions can be summed up. This procedure, of course, bears the risk that additional use cases must be supported to resolve the identified problems for specific organisations. Further research with different objectives and specific problems should be conducted to minimise this risk.

The last objective was to implement the proposed approach for Active Directory as a demonstration. This was achieved in Chapter 6 "Demonstration" and will be discussed further down in this chapter.

7.2.2 Compliance to Established Criteria

Based on the prior research and findings, we created four cornerstones of requirements in section 4.1.4 and derived four essential criteria for design choices to give us a guideline on what to consider when deciding which implementation option is best for our use case based on its set of advantages and disadvantages. We will now check whether these criteria can all be met in the proposed solution.

Criteria one demands that all data identified as necessary in section 4.1.1 must be included in the uniform structure. These four types are the department-to-department connections, the employee-to-department connections, and the master data records for employees and departments. All four of these data sets are included in the proposed final structure. While the master data records were adopted, the connection data was included in the master data dictionaries for employees and departments. The criteria are, therefore, fully met.

Based on the bidirectional use cases, the second criterion was that the uniform structure must allow for (efficient) upward and downward traversal. This is achieved by permanently storing a connection in both adjacent nodes. While employees store their immediate departments, departments store their immediate employees and their subordinate and super-ordinate departments. This allows for traversal in both directions in $O(n)$ from top to bottom or reverse. This represents the best possible time for visiting every node. The only faster way would be to store all indirect connections and jump directly to the intended target. However, we will consider the criteria successful since this was discussed in section 4.1.4 and rejected.

The third criterion was to enable the mapping of many-to-many relationships between all entities. Because every entity stores all of its connections and these storage fields are all organised assets, it must be possible for every entity to store more than one connection per connection type. Accordingly, the criteria are met.

Criteria four demands low memory utilisation and access times. This topic was intensively discussed in section 4.1.4 since all of the other three criteria could easily be achieved by all reviewed solutions. Based on the assumptions made in this section, the best option was selected. We therefore believe that the criteria were met as best as possible. Considering that only three companies were employing more than 1.000.000 people in the financial year 2022/23, it is reasonable to assume that a runtime complexity of $O(m + d)$ will result in sufficient access times for the identified problems (Statista Research Department, 2024a).

7.2.3 Compatibility Test with Real Data

In Chapter 6 tested the unified structure's compatibility with data extracted from Active Directory, representing one of the identified sources of organisational structure data. This involved mapping data fields from these systems to our uniform structure. No data loss or distortion has occurred.

While our mapping using schema_files is a safe method in theory, it still raises a larger question: Should there be any safety measures to detect incorrect or missing data before a data source can add it to a pool of unified organisation structure data? This question is an essential topic regarding practical implementations and usability and should be addressed in further research!

7.2.4 Integration Test with Existing Tools

A seamless integration with existing tools is essential for a tool to obtain practical impact. Two interfaces must be considered for our proposed solutions:

The source systems for organisational structure data and the downstream systems using the results for further applications. Regarding the access of the examined source data systems, we discussed how to import data from SAP or AD in section 5.4. We have seen that SAP is not the most straightforward system to obtain data from, while AD poses more considerable challenges.

Therefore, this side of the integration seems twofold and should be evaluated in individual cases once they occur. Regarding the downstream systems, research has yet to be conducted. However, for both connection types, the modality of the proposed solution and the large number of libraries available for Python offer a massive flexibility with which almost every data source and downstream system should be able to be connected.

7.2.5 Evaluation of the Impact on Identified Problems

The identified problems in Chapter 1 stated that developers cannot be connected to their departments. This problem is resolved by storing the immediate departments a developer belongs to.

Next the problem is presented, that an organisational must be able to sum up the contributions of developers to various levels of the organisation. This problem is presented by the second of the three use cases discussed in detail and therefore resolved.

The next two problems belong together. It is stated, that the above mentioned problems are not easily solvable because organisational structure data exists in various different forms and there is no unified structure for organisational struc-

ture data, These two problems are resolved by definition by designing a uniform structure and proposing an approach to implement this structure.

7.2.6 Summary of the results

In summary, it can be stated that the chapter outlines the successful design and implementation of a uniform structure for organisational structure data, demonstrating its practicality, compliance with established criteria, and ability to solve identified problems. Still there is a lot of room for further research at it is suggested above.

7.3 Practical Implications of the Research

Regarding the practical use of this research, the results are twofold. On the one hand, only a few ready-to-use implementation elements exist. Only the data source Active Directory can be connected now, and the results can only be stored in a JSON file. Therefore, the practical usability could be improved in its current state of implementation.

On the other hand, the current implementation served primarily as a proof of concept and was never expected to be ready to be shipped to companies for daily use. Conversely, the theoretical concept is fully developed and can be applied to any data source. This, together with the modular structure of the implementation, makes it easy for future users to use the current results as a sound basis and develop their own data connections for their specific use cases.

7.4 Limitations of the Proposed Solution

Of course, limitations are always essential to evaluating a proposed solution since it is important to know under which circumstances the results may differ from those achieved in this thesis. When it comes to the limitations of this research, there are quite a few topics that have to be discussed:

First, the technical realisation of the proposed ideal uniform structure relies on assumptions. Companies with many more departments than employees in the data system or companies where every user is assigned to more than two departments on average. The latter could be a consultancy, often organised as matrix structures. Therefore, section 4.1.4 should be re-read before the uniform structure is deployed on any new set of entities. The results depend on divergent variables, so other options can be chosen if the assumptions are incorrect for a specific use case.

A second limitation is that only digitised data with an inherent structure can be used for this uniform structure. Many companies may use an organisational structure drawn on paper or one Excel file for all relationships. In these cases, unifying can get very hard if, for example, the Excel file does not follow an inherent structure but is drawn arbitrarily.

Next, we only inspected the data structure of two systems containing organisational structure data. Many more should be inspected to uncover exceptional cases, differentiations, or limitations yet to be considered. Systems designed for specific industries might contain additional structures or specialities important to their sector. Therefore, in future research, many more systems should be examined, and further findings should be included in the uniform data structure.

A speciality to the last limitation is that no systems were inspected, focusing only on a company's Human Resources section. These systems will undoubtedly contain organisational structure data and might be the systems most specified for representing a company's structure.

Lastly, a very important limitation is that no combination of data from different systems was discussed. Employee-specific data is still a significant problem in the industry; it is scattered across multiple systems and needs to be consolidated. For this task, the uniform structure is the perfect solution. However, once data from different data sources is combined, new problems emerge, like how the data sets can be connected. Usually, every system has its own way of creating a unique ID for every entity. Matching these IDs can be tricky at times. Apart from that, the data formats might not match, e.g. the date format or the accuracy with which timestamps are specified, the contained information can be contradictory or inconsistent and much more. These familiar issues should be addressed in future research to enable combining data from different source systems.

8 Conclusion

In this thesis, we set out to address the complexity of managing organisational structure data in inner-source software development. Through careful research and development, we have developed a novel, uniform structure to standardise and unify the storage and processing of organisational data. As a result, we proposed a uniform structure and an implementation approach to address the identified problems. These artefacts were later demonstrated and evaluated. Even though several limitations were revealed, the evaluation showed that the proposed solution could solve many, if not all, of the problems identified.

Our contribution to science and business is twofold. First, we have presented a comprehensive analysis and solution to the difficulties of organising and using corporate data - a topic that, while not groundbreaking in its novelty, is a significant area of concern for many companies. Even simple tasks such as maintaining consistent data records for every employee or consolidating existing data from different systems cause problems for companies in our modern and digitised world. Our approach provides a structured methodology to mitigate this challenge and improve efficiency and clarity in handling such data.

Thinking critically about this thesis, it becomes clear that while progress was made in solving some fundamental problems, the topic of organisational data management is large and complex. While our solution is theoretically finished, it is only a step towards a more integrated and universally applicable system. The limitations identified in this work, particularly in terms of integration with existing systems and extension to different use cases, underline the need for further research and development.

The possible applications and extensions of this work are huge. Different use cases may require different design decisions, providing opportunities for future research to develop the unified structure proposed in this thesis further. Furthermore, when thinking about the applications of the uniform structure, the idea comes to mind that this system could be extended to a knowledge database about employees. With this, different tasks could be performed, from employee evaluation to analysing the hire-to-retire process. These applications are currently

impossible because it is hard to consolidate all data points for employees, who are often scattered across many systems.

This work lays the foundation for a more unified approach to managing organisational structure data. It provides insights and solutions to a problem that challenges organisations worldwide. Looking to the future, it is clear that the path to seamless integration and use of organisational data still needs to be completed.

Appendices

A Guide to Accessing the MS Graph API with Python

A.1 Creating an Access Point

1. Go to https://portal.azure.com and log in if necessary.
2. Search for "App registrations" and create a new registration at the top left.
3. Choose a name and select option 3 "Accounts in any organizational directory + personal accounts" under "Supported account types" – this is the most flexible option keeping all options open for the future.
4. The Redirect URI is only required if you want to use OAuth2; otherwise, leave it blank and confirm the registration.

A.2 Settings in the Access Point

1. You will be taken to the page of your access point. You can find this page again later by searching for "App registrations". Your access point will now be listed.
2. Several settings need to be configured. Let's start with Authentication: Select "Authentication" on the left menu, scroll down, and set both to YES under "Advanced settings". Without YES at "Allow public client flows", Python access did not work.
3. Certificates and secrets: Here, you must create a new client secret key and copy the "Value".
   WARNING: You only need the "Value", but you can see it only once at the start and cannot display it again!
4. API Permissions: In this menu, you can set what you can access with this access point. Select "Add permissions" and find out all the necessary permissions.
   WARNING: To enable access with the Python code presented here, "Application permissions" are required. We access in the background since this does not require a separate Microsoft account linked with the tenant. Therefore, "Delegated permissions" are not sufficient unlike described in 99% of the guides!
   The important permissions can be found under "Graph API" (Maximum 30 per access point!). To preview things, I recommend the Graph Explorer: https://developer.microsoft.com/en-us/graph/graph-explorer

5. After adding permissions, they must be granted by clicking "Grant admin consent for 'MSFT'". **WARNING:** If you do not have the necessary permissions, you must grant them in the Admin Portal (partly possible yourself!). Go to https://admin.microsoft.com and check under Users.

A.3 Access via Python

1. Required IDs: We stay in Azure and go to the "Overview" menu item. There we need the following IDs:
   (a) Application ID (Client): This is the ID of your access point and tells Microsoft which access point you want to use.
   (b) Directory ID (Tenant): Otherwise described in most Microsoft admin portals as "Tenant ID". It indicates which tenant you want to access.
   (c) Client Secret ID: The ID we created in point 3 of the settings under "Certificates and Secrets".
2. The Python script in the appendix is fully functional and gives you a rough idea of how you can access the Graph API. Replace the three IDs in the main method with 1a-c and run the script as a test.

A.4 Appendix: Python Script

```
import msal
import requests

def get_token(client_id, tenant_id, client_secret):
    authority = f"https://login.microsoftonline.com/{tenant_id}"
    app = msal.ConfidentialClientApplication(
        client_id, authority=authority,
        client_credential=client_secret)
    result = app.acquire_token_silent(
        ["https://graph.microsoft.com/.default"], account=None)
    if not result:
        result = app.acquire_token_for_client(
            scopes=["https://graph.microsoft.com/.default"])
    if 'access_token' in result:
        print(result['access_token'])
        return result['access_token']
    else:
        print(f"Could not acquire token: {result}")
        return None
```

```
def call_graph_api(token):
    graph_endpoint = 'https://graph.microsoft.com/v1.0/users'
    headers = {
        'Authorization': f'Bearer {token}',
        'Accept': 'application/json',
        'Content-Type': 'application/json',
    }
    response = requests.get(graph_endpoint, headers=headers)
    if response.status_code == 200:
        return response.json()
    else:
        print(f"Graph API call failed: {response}")
        return None

def main():
    client_id = 'YOUR CLIENT ID'
    tenant_id = 'YOUR TENANT ID'
    client_secret = 'YOUR SECRET CLIENT ID'
    token = get_token(client_id, tenant_id, client_secret)
    if token:
        data = call_graph_api(token)
        print(data)

if __name__ == "__main__":
    main()
```

A.5 Appendix: Error Handling

This section briefly addresses the errors encountered:

1. HTTP Status Code 400 "Bad Request" indicates the server has interpreted your request as invalid and cannot process it. This is likely an external error. Check the following possible causes:

   (a) Ensure you are calling the correct endpoint. The endpoint 'https://graph.microsoft.com/v1.0/me' returns information about the currently authenticated user. However, if you are using the client credentials flow, there may not be a "current user". Try calling an endpoint that does not require user interaction, such as 'https://graph.microsoft.com/v1.0/users'.

   (b) Check if your app registration in Azure has the necessary permissions to call the Graph API. Ensure you have granted the correct application permissions in Azure and assigned these permissions.

   (c) Ensure the access token you are using is valid. You might want to print the token and verify it with a JWT decoder.

2. HTTP Status Code 403 "Forbidden" means the server understands your request but refuses it. With the Microsoft Graph API, this usually occurs when you do not have the required permissions to access the requested resource. Check the following three steps:

   (a) App permissions: The access point does not have the necessary access rights → Refer to "Settings Item 4".

   (b) Token has requested the wrong permissions → Refer to "Settings Item 4 CAUTION" → '/.default' always accesses everything that has been released. If you use something different, you can use jwt.io or jwt.ms to analyze your token.

   (c) Other causes: Requested resource (in request.get() in call_graph_api()), API version (the v1.0 in graph_endpoint in call_graph_api()).

B Results for Data Extract from Active Directory Demonstration System

To allow a more detailed examination of the Active Directly data structure, the following appendix contains the results from our demonstration system.

B.1 Results for Users of the Organisation

This is only a selection of the results. The selection was made because further results didn't contain additional information about the data structure.


```
{
    "@odata.context": "https://graph.microsoft.com/v1.0/$metadata#users",
    "value": [
        {
            "businessPhones": [
                "8006427676"
            ],
            "displayName": "MOD Administrator",
            "givenName": "MOD",
            "jobTitle": null,
            "mail": "admin@M365x214355.onmicrosoft.com",
            "mobilePhone": "5555555555",
            "officeLocation": null,
            "preferredLanguage": "en-US",
            "surname": "Administrator",
            "userPrincipalName": "admin@M365x214355.onmicrosoft.com",
            "id": "5bde3e51-d13b-4db1-9948-fe4b109d11a7"
        },
        {
            "businessPhones": [
                "+1 858 555 0110"
            ],
            "displayName": "Alex Wilber",
            "givenName": "Alex",
            "jobTitle": "Marketing Assistant",
            "mail": "AlexW@M365x214355.onmicrosoft.com",
            "mobilePhone": null,
            "officeLocation": "131/1104",
            "preferredLanguage": "en-US",
            "surname": "Wilber",
```

```
        "userPrincipalName": "AlexW@M365x214355.onmicrosoft.com",
        "id": "4782e723-f4f4-4af3-a76e-25e3bab0d896"
    },
    {
        "businessPhones": [
            "+1 262 555 0106"
        ],
        "displayName": "Allan Deyoung",
        "givenName": "Allan",
        "jobTitle": "Corporate Security Officer",
        "mail": "AllanD@M365x214355.onmicrosoft.com",
        "mobilePhone": null,
        "officeLocation": "24/1106",
        "preferredLanguage": "en-US",
        "surname": "Deyoung",
        "userPrincipalName": "AllanD@M365x214355.onmicrosoft.com",
        "id": "c03e6eaa-b6ab-46d7-905b-73ec7ea1f755"
    },
    {
        "businessPhones": [],
        "displayName": "Conf Room Baker",
        "givenName": null,
        "jobTitle": null,
        "mail": "Baker@M365x214355.onmicrosoft.com",
        "mobilePhone": null,
        "officeLocation": null,
        "preferredLanguage": null,
        "surname": null,
        "userPrincipalName": "Baker@M365x214355.onmicrosoft.com",
        "id": "013b7b1b-5411-4e6e-bdc9-c4790dae1051"
    },
    {
        "businessPhones": [
            "+1 732 555 0102"
        ],
        "displayName": "Ben Walters",
        "givenName": "Ben",
        "jobTitle": "VP Sales",
        "mail": "BenW@M365x214355.onmicrosoft.com",
        "mobilePhone": null,
        "officeLocation": "19/3123",
        "preferredLanguage": "en-US",
        "surname": "Walters",
```

```
        "userPrincipalName": "BenW@M365x214355.onmicrosoft.com",
        "id": "f5289423-7233-4d60-831a-fe107a8551cc"
    },
    {
        "businessPhones": [],
        "displayName": "Brian Johnson (TAILSPIN)",
        "givenName": "Brian",
        "jobTitle": null,
        "mail": "BrianJ@M365x214355.onmicrosoft.com",
        "mobilePhone": null,
        "officeLocation": null,
        "preferredLanguage": null,
        "surname": "Johnson",
        "userPrincipalName": "BrianJ@M365x214355.onmicrosoft.com",
        "id": "e46ba1a2-59e7-4019-b0fa-b940053e0e30"
    },
    {
        "businessPhones": [
            "+1 858 555 0111"
        ],
        "displayName": "Christie Cline",
        "givenName": "Christie",
        "jobTitle": "Sr. VP Sales & Marketing",
        "mail": "ChristieC@M365x214355.onmicrosoft.com",
        "mobilePhone": null,
        "officeLocation": "131/2105",
        "preferredLanguage": "en-US",
        "surname": "Cline",
        "userPrincipalName": "ChristieC@M365x214355.onmicrosoft.com",
        "id": "b66ecf79-a093-4d51-86e0-efcc4531f37a"
    },
    {
        "businessPhones": [],
        "displayName": "Conf Room Crystal",
        "givenName": null,
        "jobTitle": null,
        "mail": "Crystal@M365x214355.onmicrosoft.com",
        "mobilePhone": null,
        "officeLocation": null,
        "preferredLanguage": null,
        "surname": null,
        "userPrincipalName": "Crystal@M365x214355.onmicrosoft.com",
        "id": "8528d6e9-dce3-45d1-85d4-d2db5f738a9f"
```

```
        },
        {
            "businessPhones": [
                "+1 425 555 0105"
            ],
            "displayName": "Debra Berger",
            "givenName": "Debra",
            "jobTitle": "Administrative Assistant",
            "mail": "DebraB@M365x214355.onmicrosoft.com",
            "mobilePhone": null,
            "officeLocation": "18/2107",
            "preferredLanguage": "en-US",
            "surname": "Berger",
            "userPrincipalName": "DebraB@M365x214355.onmicrosoft.com",
            "id": "d4957c9d-869e-4364-830c-d0c95be72738"
        },
        {
            "businessPhones": [
                "+1 205 555 0108"
            ],
            "displayName": "Diego Siciliani",
            "givenName": "Diego",
            "jobTitle": "CVP Finance",
            "mail": "DiegoS@M365x214355.onmicrosoft.com",
            "mobilePhone": null,
            "officeLocation": "14/1108",
            "preferredLanguage": "en-US",
            "surname": "Siciliani",
            "userPrincipalName": "DiegoS@M365x214355.onmicrosoft.com",
            "id": "24fcbca3-c3e2-48bf-9ffc-c7f81b81483d"
        }
    ]
}
```


B.2 Results for All Users of the Organisation

This is only a selection of the results. The selection was made because further results didn't contain additional information about the data structure.


```
"value": [
    {
        "id": "02bd9fd6-8f93-4758-87c3-1fb73740a315",
```

```
    "deletedDateTime": null,
    "classification": null,
    "createdDateTime": "2017-07-31T18:56:16Z",
    "creationOptions": [
        "ExchangeProvisioningFlags:481"
    ],
    "description": "Welcome to the HR Taskforce team.",
    "displayName": "HR Taskforce",
    "expirationDateTime": null,
    "groupTypes": [
        "Unified"
    ],
    "isAssignableToRole": null,
    "mail": "HRTaskforce@M365x214355.onmicrosoft.com",
    "mailEnabled": true,
    "mailNickname": "HRTaskforce",
    "membershipRule": null,
    "membershipRuleProcessingState": null,
    "onPremisesDomainName": null,
    "onPremisesLastSyncDateTime": null,
    "onPremisesNetBiosName": null,
    "onPremisesSamAccountName": null,
    "onPremisesSecurityIdentifier": null,
    "onPremisesSyncEnabled": null,
    "preferredDataLocation": null,
    "preferredLanguage": null,
    "proxyAddresses": [
        "SMTP:HRTaskforce@M365x214355.onmicrosoft.com",
        "SPO:SPO_896cf652-b200-4b74-8111-c013f64406cf@SPO_dcd219dd-bc68-4b9b-bf0b-
    ],
    "renewedDateTime": "2020-01-24T19:01:14Z",
    "resourceBehaviorOptions": [],
    "resourceProvisioningOptions": [
        "Team"
    ],
    "securityEnabled": false,
    "securityIdentifier": "S-1-12-1-45981654-1196986259-3072312199-363020343",
    "theme": null,
    "visibility": "Private",
    "onPremisesProvisioningErrors": [],
    "serviceProvisioningErrors": []
},
{
```

```
        "id": "06f62f70-9827-4e6e-93ef-8e0f2d9b7b23",
        "deletedDateTime": null,
        "classification": null,
        "createdDateTime": "2017-07-31T17:38:15Z",
        "creationOptions": [],
        "description": "Video Production",
        "displayName": "Video Production",
        "expirationDateTime": null,
        "groupTypes": [
            "Unified"
        ],
        "isAssignableToRole": null,
        "mail": "VideoProduction@M365x214355.onmicrosoft.com",
        "mailEnabled": true,
        "mailNickname": "VideoProduction",
        "membershipRule": null,
        "membershipRuleProcessingState": null,
        "onPremisesDomainName": null,
        "onPremisesLastSyncDateTime": null,
        "onPremisesNetBiosName": null,
        "onPremisesSamAccountName": null,
        "onPremisesSecurityIdentifier": null,
        "onPremisesSyncEnabled": null,
        "preferredDataLocation": null,
        "preferredLanguage": null,
        "proxyAddresses": [
            "SMTP:VideoProduction@M365x214355.onmicrosoft.com",
            "SPO:SPO_16219fd2-fafd-4fea-8084-8b5eaa8c5ad2@SPO_dcd219dd-bc68-4b9b-bf
        ],
        "renewedDateTime": "2017-07-31T17:38:15Z",
        "resourceBehaviorOptions": [],
        "resourceProvisioningOptions": [],
        "securityEnabled": true,
        "securityIdentifier": "S-1-12-1-116797296-1315870759-261025683-595303213",
        "theme": null,
        "visibility": "Public",
        "onPremisesProvisioningErrors": [],
        "serviceProvisioningErrors": []
    },
    {
        "id": "0a53828f-36c9-44c3-be3d-99a7fce977ac",
        "deletedDateTime": null,
        "classification": null,
```

```
        "createdDateTime": "2017-09-02T02:54:25Z",
        "creationOptions": [
            "YammerProvisioning"
        ],
        "description": "Marketing Campaigns",
        "displayName": "Marketing Campaigns",
        "expirationDateTime": null,
        "groupTypes": [
            "Unified"
        ],
        "isAssignableToRole": null,
        "mail": "marketingcampaigns@M365x214355.onmicrosoft.com",
        "mailEnabled": true,
        "mailNickname": "marketingcampaigns",
        "membershipRule": null,
        "membershipRuleProcessingState": null,
        "onPremisesDomainName": null,
        "onPremisesLastSyncDateTime": null,
        "onPremisesNetBiosName": null,
        "onPremisesSamAccountName": null,
        "onPremisesSecurityIdentifier": null,
        "onPremisesSyncEnabled": null,
        "preferredDataLocation": null,
        "preferredLanguage": null,
        "proxyAddresses": [
            "SMTP:marketingcampaigns@M365x214355.onmicrosoft.com",
            "SPO:SPO_8cfbec68-642c-4d90-a15e-68d5d55e1c1f@SPO_dcd219dd-bc68-4b9b-bf0b-
        ],
        "renewedDateTime": "2017-09-02T02:54:25Z",
        "resourceBehaviorOptions": [
            "YammerProvisioning"
        ],
        "resourceProvisioningOptions": [],
        "securityEnabled": false,
        "securityIdentifier": "S-1-12-1-173245071-1153644233-2811837886-2893539836",
        "theme": null,
        "visibility": "Public",
        "onPremisesProvisioningErrors": [],
        "serviceProvisioningErrors": []
    }
]
```

References

Buchner, R. D., S. (2022). Calculating the costs of inner source collaboration by computing the time worked. *SAIS 2004 Proceedings.*, *47*. https://aisel.aisnet.org/sais2004/47

Buchner, R. D., S. (2023). A research model for the economic assessment of inner source software development. *SAIS 2004 Proceedings.*, *47*. https://aisel.aisnet.org/sais2004/47

Capraro, M., & Riehle, D. (2016). Inner source definition, benefits, and challenges. *ACM Computing Surveys*, *49*, 1–36. https://doi.org/10.1145/2856821

Carroll, N., Morgan, L., & Conboy, K. (2018). Examining the impact of adopting inner source software practices, 1–7. https://doi.org/10.1145/3233391.3233530

Cooper, D., & Stol, K.-J. (2018). *Adopting innersource: Principles and case studies.*

Gurbani, V. K., Garvert, A., & Herbsleb, J. D. (2010). Managing a corporate open source software asset. *Commun. ACM*, *53*(2), 155–159. https://doi.org/10.1145/1646353.1646392

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*, 45–77.

Statista Research Department. (2024a). Größte unternehmen weltweit nach anzahl der beschäftigten 2022/2023 [Zugriff am 01. März 2024].

Statista Research Department. (2024b). *Marktanteile der anbieter am umsatz mit crm-software weltweit 2022*. Retrieved March 1, 2024, from https://de.statista.com/statistik/daten/studie/262328/umfrage/marktanteile-der-anbieter-von-crm-software-weltweit/

Stol, K.-J., Avgeriou, P., Babar, M. A., Lucas, Y., & Fitzgerald, B. (2014). Key factors for adopting inner source. *ACM Trans. Softw. Eng. Methodol.*, *23*(2). https://doi.org/10.1145/2533685