Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik

Course of study Industry 4.0 Informatics

Bachelor Thesis

by

Adrian **Rall**

Konzeption und Entwicklung eines Dependency-Update-Mechanismus zur automatischen Distribution neuer Software-Versionen im Sondermaschinen- und Anlagenbau

Conception and Development of a Dependency Update Mechanism for the Automatic Distribution of a New Software Version in Special Machinery and Industrial Engineering

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik

Course of study Industry 4.0 Informatics

Bachelor Thesis

by

Adrian **Rall**

Konzeption und Entwicklung eines Dependency-Update-Mechanismus zur automatischen Distribution neuer Software-Versionen im Sondermaschinen- und Anlagenbau

Conception and Development of a Dependency Update Mechanism for the Automatic Distribution of a New Software Version in Special Machinery and Industrial Engineering

Processing time:	from	November 2nd, 2022
	to	April 28th, 2023

First Examiner: Prof. Dr.-Ing. Christoph P. Neumann

Second Examiner: Prof. Dr. rer. nat. Kurt Hoffmann

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik

Confirmation according to § 12 APO

Name and first name
of the student: **Rall, Adrian**

Course of studies: **Industry 4.0 Informatics**

I confirm that I have independently written the Bachelor Thesis entitled:

Conception and Development of a Dependency Update Mechanism for the Automatic Distribution of a New Software Version in Special Machinery and Industrial Engineering

have not submitted it for examination elsewhere, have used no sources or aids other than those indicated, and have identified any verbatim or analogous citations as such.

Date: April 26, 2023

Signature:

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektro- und Informationstechnik

Bachelor Thesis Zusammenfassung

Student (Name, Prename):	**Rall, Adrian**
Course of study:	Industry 4.0 Informatics
Supervisor:	Prof. Dr.-Ing. Christoph P. Neumann
Carried out at (company/authority/university):	BHS-Corrugated Anlagen- und Maschinenbau GmBH
Supervisor of the Company:	Benedikt Bräutigam
Date of issue: November 2nd, 2022	*Deadline* : *April*28th, 2023

Title:

Conception and Development of a Dependency Update Mechanism for the Automatic Distribution of a New Software Version in Special Machinery and Industrial Engineering

Abstract:

This thesis presents the design and implementation of an automated version update mechanism for npm and NuGet packages and Github submodules in an organizational-wide repository structure. The mechanism improves an existing but inefficient process by considering the dependencies of packages and submodules to optimize storage, management, and handling of the update mechanism. The design and development of the mechanism are tailored organizational wide enterprise server but are easily extendable to other repository structures. The procedure is implemented in two steps in the overall process: first, all relevant data are collected and stored; second, a recursive depth-first search with a topological sorting algorithm is performed for directed acyclic graphs. The result can be integrated into the business workflow.

Keywords: Dependency update mechanism, Reusable Software Parts, object-oriented project management, github actions, automation, workflow integration

Contents

1 Context 1
1.1 Motivation 1
1.2 Context of this thesis 2
1.3 Objective 2

2 BHS Corrugated 3
2.1 Introduction of the company 3
2.2 BoxPlant 2025 - upgrade as a service 4
2.3 iCorr - flagship for digitalization 4
2.4 Conclusion 4

3 Questionnaire 6
3.1 Quality criteria 6
3.2 Hypotheses 7
3.3 Results questionnaire 7
3.4 Conclusion 8

4 Usage and embedding 10
4.1 Integration into the company 10
4.1.1 CI/CD 10
4.1.2 Agile V-model 11
4.2 Old Dependency-Update mechanism 14
4.2.1 Background of the Old Mechanism 14
4.2.2 Features of the old mechanism 14
4.3 Weaknesses and limitations of the old mechanism 15
4.4 Cost savings with improved dependency update mechanism 15
4.5 Criticism of the old mechanism 16
4.6 Benefits due to the mechanism 16

5 Literature overview and state of the art 17
5.1 DevBots 17
5.1.1 Dependabot 18
5.1.2 Renovate 18
5.1.3 Greenkeeper 18

5.1.4 Further Alternatives . . . 18
5.2 Limitations and problems . . . 19
5.2.1 Companies experiences with Dependabot . . . 20
5.3 Advantages of external bots . . . 21
5.4 Conclusion . . . 21

6 Methodology 22
6.1 Purpose . . . 22
6.2 Background information . . . 22
6.2.1 NPM . . . 22
6.2.2 NuGet . . . 23
6.2.3 Submodules . . . 23
6.3 JSON format for relevant information . . . 23
6.4 Directed acyclic graph . . . 26
6.4.1 Definition . . . 26
6.4.2 Complexity . . . 26
6.4.3 Depth-first Search . . . 27
6.4.4 Breath-first search . . . 27
6.4.5 Cycle detection . . . 27
6.4.6 Topological Sorting . . . 29
6.5 Conclusion . . . 29

7 Implementation 30
7.1 Description of the implemented solution . . . 30
7.2 Action No. 1: Write2Inventory . . . 31
7.2.1 Purpose of the Action . . . 31
7.2.2 Code and functionality . . . 31
7.2.3 Conclusion . . . 34
7.3 Action No. 2: “DependencyUpdate” . . . 36
7.3.1 Purpose of the Action . . . 36
7.3.2 Code and functionality . . . 36
7.4 Challenges and limitations . . . 42
7.4.1 Challenges and limitations of “Write2Inventory” . . . 42
7.4.2 Challenges and limitations of DependencyUpdate . . . 43
7.5 Conclusion . . . 43

8 Experimental Results 44
8.1 Test data and methodology . . . 44
8.1.1 Test applications for Write2Inventory . . . 44
8.1.2 Test applications for “DependencyUpdate” . . . 45
8.2 Analyse und Diskussion der Ergebnisse . . . 45

9 Summary and future work 46
9.1 Summary of the results . . . 46
9.2 Future work . . . 46

Literaturverzeichnis **48**

Abbildungsverzeichnis **51**

Tabellenverzeichnis **52**

A Rohdaten **54**
A.1 Statistical methods 54
A.2 Statistical evaluation of the questionnaire 54
A.2.1 Question (F10) Push-Pull-cycles 55
A.3 Example output of Write2Inventory 57
A.4 Whole workflow 59

Symbole, Formelzeichen und Einheiten

Chapter 1

Context

1.1 Motivation

According to the statistics of the Federal Employment Agency [4], [5], the number of unfilled positions of STEM professions (professions from the fields of Science, Technology, Engineering, and Mathematics) in Germany has been 263,000 in 2019. In 2018, an average of 66% german companies had difficulties filling positions advertised specifically for IT professionals.

To counteract te shortage of skilled workers, one option is to outsource routine programming tasks to offshore IT services. Corporate IT departments are thus given the opportunity to focus on creating relevant program sections where company-specific knowledge is needed. This system of management has a project-related, but also company-related aspect.

Over time, an increasing number of projects develop with different software components and versions. Automating the update management reduces the time and effort required to keep systems and associated documentation up to date.

Kula et al. (2018) [21] analyzed whether users regularly update third-party packages that are included in their code are updated to the latest version. The results suggest that users rarely update their libraries. Reasons are said to be the additional workload, a lack of motivation, or the incorrect assessment of the risk of compromise.

McDonnell et al. (2013) [7] state that customer acceptance with the rapid revision and publication cycles of third-party embedding has a negative impact on the constant updating of the external code.

In Addition, Bogart et al. (2015) [8] mention in their surveys, that "non-technical forms of organization" [21], such as workload and especially lack of understanding of the responsibilities of developers play a decisive role in whether updates are migrated or not. A habituation effect and the associated trust in a particular library version resulting in a lack of willingness to incorporate newer versions, also play a role, even if a trend toward shorter latency is discernible according to the study.

However, it is not only the reduced workload of the employees that must be taken into consideration, but also the minimization of security-related bugs.

According to the German Federal Office for Information Security (BSI), "10% more vulnerabilities in software products became known in 2021 than in the previous year. More than half of them had high or critical scores according to the Common Vulnerability Scoring System (CVSS)" [13]. It is said that 13% of the vulnerabilities alone were in the critical range.

The research of Bier et al. (2021) [9] for example indicates a rather small number of attachment points for a broad fix of "Remote Code Execution" (RCE) vulnerabilities.

1.2 Context of this thesis

This paper is being carried out in the Digital Engineering program of the company BHS Corrugated Maschinen- und Anlagenbau GmbH.

In 2021, the Digital Engineering program moved from ... to Github Enterprise as a coordination platform. The department's scope of work lies in the coordination of mechanics, electronics and software development. For this purpose, a specially structured ... was created to keep code sections as modular and reusable as possible. A historically grown but inefficient dependency update mechanism is to be replaced by an efficient solution without losing effectiveness.

1.3 Objective

The main goal of the study is the conception and implementation of a dependency update mechanism. The primary objective is to keep NPM packages, NuGet packages and submodules up to date automatically and without manual intervention throughout the department. A simple extensibility should be possible for further packages of other package manager systems (for example Maven). Compared to the current dependency update mechanism, the solution should have

- improved efficiency
- consistent or improved effectiveness
- consistently low or even lower error-proneness
- reduce costs for pipeline usage
- improved traceability for developers

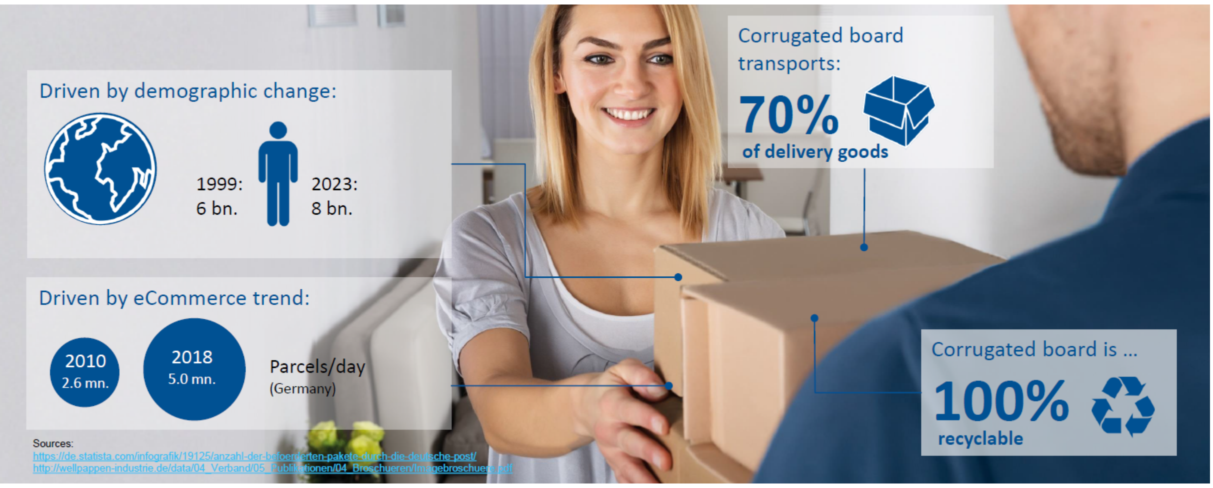
Furthermore, the departmental structure for modularity and reusability of code sections will be evaluated.

Chapter 2

BHS Corrugated

2.1 Introduction of the company

BHS Corrugated is "a global supplier of mechanical engineering, plant engineering, lifecycle service and digital solutions for the complex requirements of the corrugated board industry" [1]."With more than 3,000 employees worldwide, the company is represented in over 20 countries, not counting [the] headquarters in Weiherhammer, Germany" [2].



Sources:
https://de.statista.com/infografik/19125/anzahl-der-befoerderten-pakete-durch-die-deutsche-post/
http://wellpappen-industrie.de/data/04_Verband/05_Publikationen/04_Broschueren/Imagebroschuere.pdf

Figure 2.1: Bhs corporate statistics

BHS has been founded in 1717 as a hammer mill. With 1960 ongoing, the company transformed into the leading supplier of corrugators world wide. Nowadays the company strives for a transformation from a classic mechanical engineering company to an overall Industry 4.0-capable company. With this, an improvement in process automation, process optimization, and rising production efficiency as well as improvements of service, towarding digitalization, customers of BHS will be prepared for their future work.

BHS offers a complete range of corrugated board production solutions, from individual machines to entire production lines, from corrugating rolls, customer support, maintenance and repair to automated intralogistics. This wide range of products and strategies such as allocating 5% of annual sales to research and development of new technologies have empowered BHS Corrugated to grow into the world market leader for corrugators. With more than 50% market share of corrugators, 70% of all cardboard boxes worldwide originate from BHS machines.

Not least due to the ever increasing networking and digitalization, BHS Corrugated regularly wins prizes and honors such as the "TOP 100" award, "German Brand Award" or "Best Managed Companies Award".

2.2 BoxPlant 2025 - upgrade as a service

BoxPlant 2025 intends the company's visions and goals for 2025 and beyond. This is intended to make the future image of steadily increasing automation and autonomization concrete and tangible. This includes the provision of digital twins, the automation of intralogistics, improved modularization of plants, digital solutions for the corrugator or the digital inline printing.

2.3 iCorr - flagship for digitalization

The department for the support and development of the iCorr product family is considered the showcase model for digitalization at BHS Corrugated. While GitHub has only been adopted on a company wide basis in 2021, the department Digital Solutions (DS) around iCorr has been using GitHub since 2017. Further, most iCorr departments are already fulfilling their minimum target requirements of the BoxPlant 2025 agenda.

2.4 Conclusion

"Staufen AG has been collecting the German Industry 4.0 Index since 2014. [...] The latest survey from 2022 concludes that only every other company in Germany is actively engaged with the 'factory of the future'. In "only seven percent of companies, the future project Industry 4.0 is already comprehensively integrated into daily operations in the factory halls and development departments" [3].

Business Units such as DS already demonstrate a high level of digitalization. However, the majority of the company is still in the process of transforming itself into a highly digitalized and networked 'factory of the future'.

The company seems to be split in two, similar to the study of Staufen AG. There are departments that always use the latest digital technology. On the flip side, it appears that a large number of departments are taking their first steps in networking and digitization.

In order to have a quantitative picture of the use of tools and the degree of awareness of certain programming paradigms, a questionnaire has been sent to two departments for clarifing the situation.

Chapter 3

Questionnaire

The questionnaire is intended to reflect the current state of technology and work processes (November/December 2022), to prove certain hypotheses on the degree of object-oriented programming, and the implementation and understanding of semantic versioning with the help of the survey.

Both inductive and deductive approaches have been used within the survey. Initially, hypotheses have been formulated and tested through the survey (deductive approach). In addition, new insights have been gained through the evaluation of the survey. These have been used to create a quantitative image of the work environment and work processes and to generate new conclusions (inductive approach).

3.1 Quality criteria

The main quality criteria "validity," "reliability," and "objectivity" for the questionnaire have been considered and implemented as follows:

Definition 3.1.1 (Validity) *Himme (2007) writes: "as part of the validity check, it is necessary to ask whether the instrument measures what is supposed to measure (translated from German)" (original quote: "Im Rahmen der Validitätsprüfung ist zu fragen, ob mit einem Messinstrument das gemessen wird, was gemessen werden soll.") ([39])*
Definition 3.1.2 (Reliability) *"The criterion refers to the question how the measurement is made and demands that the measurement results should be reproducible upon repeated measurement (translated from German)" (original quote: "Das Kriterium bezieht sich auf die Frage, wie gemessen wird, und fordert, dass die Messergebnisse bei wiederholter Messung reproduzierbar sein sollten.") ([39])*
Definition 3.1.3 (Objectivity) *"Objective measurement results are obtained when different persons who carry out the measurements independently get the same measurement results (translated from German)" (original quote: "Objektive Messergebnisse liegen vor, wenn verschiedene Personen, die die Messungen unabhängig voneinander vornehmen, zu den gleichen Messergebnissen gelangen.") ([39])*

The objectivity of execution (the influence of the interviewer and the research environment on the respondents) has been ensured through standardized questioning, voluntary participation, anonymity of participants, and absence of the surveyor (participants have been able to answer the survey online and in their own chosen time period without a supervisor).

Objectivity of evaluation has been given numerical scales within the questionnaire through a statistical analysis. Only comments haven't been not predetermined. They could be submitted in free language form for each question.

3.2 Hypotheses

The following hypotheses have been investigated:
Hypothesis 1 H_0: *There is no significant correlation between the sub-questions regarding the implementation of object orientation*

This thesis has been verified using a "7-point Likert scale" in conjunction with thirteen questions. The survey has been a self-assessment. No other quantitative measurement methods have been used for this purpose. In total, the questions were answered by 15 people.

Hypothesis 2 H_0: *There is no significant difference in the frequency distributions between category 2 + 3 + 4 and category 1 for question 13.*

It has been ascertained which preferences employees have with regard to the semantic notation for the inclusion of foreign libraries/packages. We can determine how much users trust the libraries or packages they use based on this.

Hypothesis 3 H_0: *The employees of the company do not significantly use the main branch for adding features*

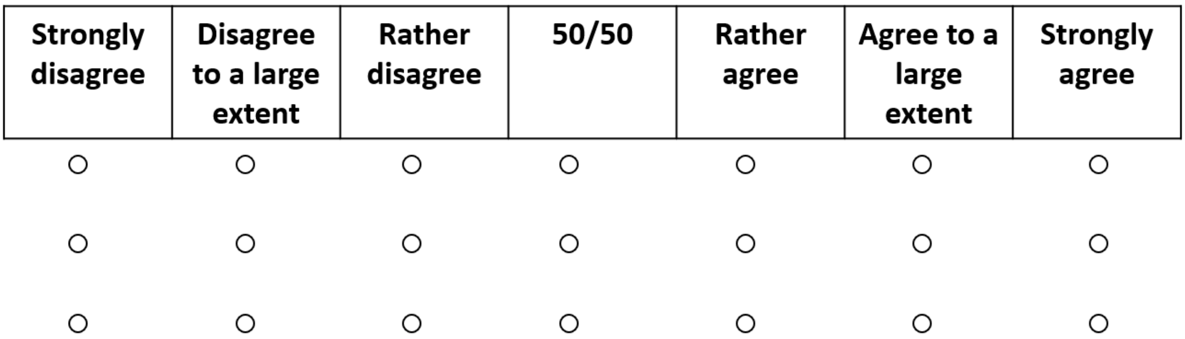
With this hypothesis, it is to be tested whether employees use the exact meaning of semantic versioning for programming features. A "minor" update often corresponds to adding new features, while a "patch" update in semantic versioning corresponds to fixing bugs. In addition, conclusions can be drawn about the frequency of the upload behavior of the main branch. The dependency update mechanism should be activated when changes are made to the main branch.

3.3 Results questionnaire

As the survey revealed, GitHub Enterprise is not yet being used throughout the company. However, the departments which already have access to the Github Enterprise Server work with Github Action for the 'Continuous Integration / Continuous Deployment' cycle (CI/CD).

The detailed statistical evaluation items of the questionnaire can be found in the appendix (A)

Hypothesis 1 has been examined with 13 questions in the form of a 7-point Likert scale. Six of the total 13 questions have been relevant with regard to the degree of implementation of the object-oriented principles.



Strongly disagree	Disagree to a large extent	Rather disagree	50/50	Rather agree	Agree to a large extent	Strongly agree
○	○	○	○	○	○	○
○	○	○	○	○	○	○
○	○	○	○	○	○	○

Figure 3.1: 7-point Likert scale

The sub-questions specifically aimed at the use of object-oriented principles. There is a strong correlation. Enough participants have answered. The statistical test can be considered valid and the null hypothesis can be rejected. In combination with the examination of significant correlations regarding sub-questions of No. 3 and No. 13, it is noted that employees of the company submit code to the main branch depending on project progress, not on a specific time frame. Nevertheless, significant correlations have been shown to exist between "I create simple but meaningful comments" and "Most of my classes have only one responsibility" with the question "How often does your code get merged to the MAIN branch (on average)?".

Hypothesis 2 could not be rejected due to the small number of questions answered. However, there is a significantly strong correlation between naming of branches and the preference for package version selection. This suggests that employees adopt versioning principles and implement them in other areas of their work environment.

Hypothesis 3 could also not be rejected due to the small number of answers. Nevertheless, a clear preference for using sub-branches is evident.

The survey results support the hypothesis that small program units designed according to object-oriented principles are being created in sub-branches. The sub-branches are named according to their functionality, whether they represent an extension of functionality ("feature" and consequently a "minor update"), or the correction of functional errors ("bugs" and consequently a "patch update"), and merged into the main branch after completion.

3.4 Conclusion

The assumption that departments of the company dealing with software production do not use the targeted state of the art has been supported by the survey. The fact that an entire department that participated in the survey does not have access to the company's Github Enterprise Server shows varying degrees of networking and digitization. The division of the German economy with regard to Industry 4.0, as described in the Staufen AG study, is also reflected within the company.

Therefore, automating dependencies in the background, without the involvement of developers, will further facilitate the handling of the platform for departments and employees who will interact with it in the future. This serves to bring other departments up to the level of the more advanced sections as quickly as possible.

Based on the survey, specifications and automation strategies were derived, which have been included in this bachelor thesis.

1. There is a basic understanding of semantic versioning.
2. A basic understanding and implementation of object-oriented working methods is available.
3. Work is generally carried out in small units.
4. There is no unified programming language, ergo no heterogeneous IDE working environment, even among departments. It follows that the dependency update mechanism must deal with packages from different management systems (NPM for node.js, NuGet for C#).

Chapter 4

Usage and embedding

This chapter is dedicated to embed the work in concrete context of usage. This chapter explores the history of the current update mechanism, how it is used and why it needs to be revised.

4.1 Integration into the company

4.1.1 CI/CD

"Continuous Integration, Delivery, and Deployment (CI/CD) is a software development practice where developers frequently integrate their code changes into a central repository. which is followed by automatic builds, tests, and deployments that verify the changes, and, if successful, deliver them to the production environment." ([52]).

During the CI phase, code changes from multiple developers are operated and managed in a shared repository. Integration can occur multiple times a day and is accompanied by automated tests to ensure that the changes do not lead to conflicts or errors in the system. This practice improves both code quality and time to release software. (Compare [52])

"Continuous Deployment (CD) is a practice of automatically releasing code changes into the production environment after passing through the CI/CD pipeline. The CD pipeline automatically deploys the successful build to the production environment, providing a faster feedback loop to the end-users." ([52])

“Continuous Delivery” (CD) and “Continuous Deployment” are often used synonymously. However, “Continuous Deployment” describes the process of automatically deploying changes to production without human intervention. In “Continuous Delivery” however, changes are deployed automatically, but the release is manual. (Compare [56])

The dependency update mechanism is being used in the integration and delivery phase of a project.

4.1.2 Agile V-model

Background

iCorr products provide services to support the use of the company's products. These products are designed to improve service performance. However, the products are not critical to the operation of a corrugator; they simply enhance the customer's comfort level. Departments around iCorr products can and do work in an agile context. All programs and modules created, culminate in a few fixed end products.

Digital engineering however forms the basis and a test environment for the transformation of the inventory from a mechatronic perspective. The final product is the digital twin. However, the modules can create added value for customers in the form of further products and modifications. At present, however, the digital twin is only used internally. Nevertheless, certain modules of the Digital Engineering program are incorporated into the development of end products. These modules, in turn, may exist in multiple variations, as they have been adapted slightly to meet the needs of different customers. In addition, these variations can have different versions (see Figure 4.1).

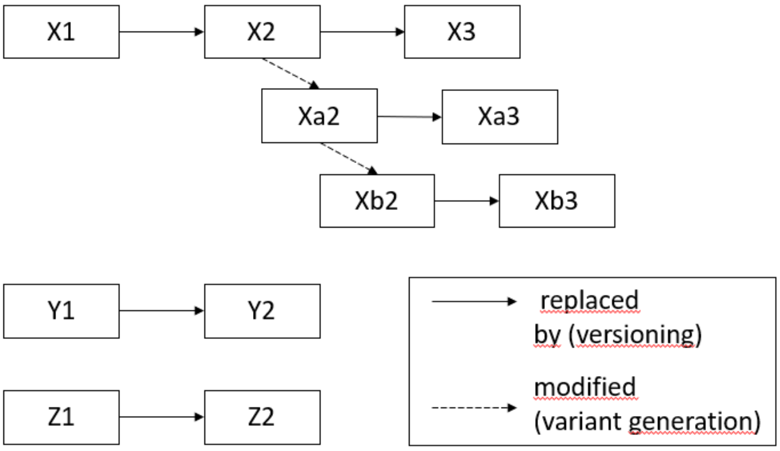


Figure 4.1: Development of versions and variants

(aus Quelle p. 555 [55] übernommen)

Illustration of the agile V-model

The work of the Digital Engineering program covers mechanics, electronics, and software development. Examples of the scope of work are MQTT communication, OPC UA data exchange, programming of the controllers (PLC), or the cyberphysical realtime simulations of a corrugator.

A classic agile working style hasn't been implemented. Instead, a combination of V-model and agile development has been developed (see Figure 4.2). Following the strict hierarchy of the V-model the individual components are assigned to one of the levels “Basics”, “Components”, “Technologies”, or “Modules”. The names of the hierarchy levels originate from the three sectors of mechanics, electronics, and software development.

Adapting the advantages of object-oriented programming

The agile V-model aims, based on object-oriented programming, to be modular (encapsulation), flexible (polymorphism), secure (encapsulation and abstraction), easily extensible and reusable (inheritance). Functions and code fragments of a lower level are only released to the next higher level, when specified **safety** and quality characteristics requirements have been met(inspired by the verification and validation principles of the V-model). Code fragments of one level can then access and use units of the same or the next lower level. This approach enables **modularity** and **reusability** of code fragments for all developers and allows for an agile way of working with the provided elements, while maintaining the quality of the classical V-Model approach.

The strict subdivision in the hierarchy levels and the respective test procedures are intended to guarantee the most error-free process possible.

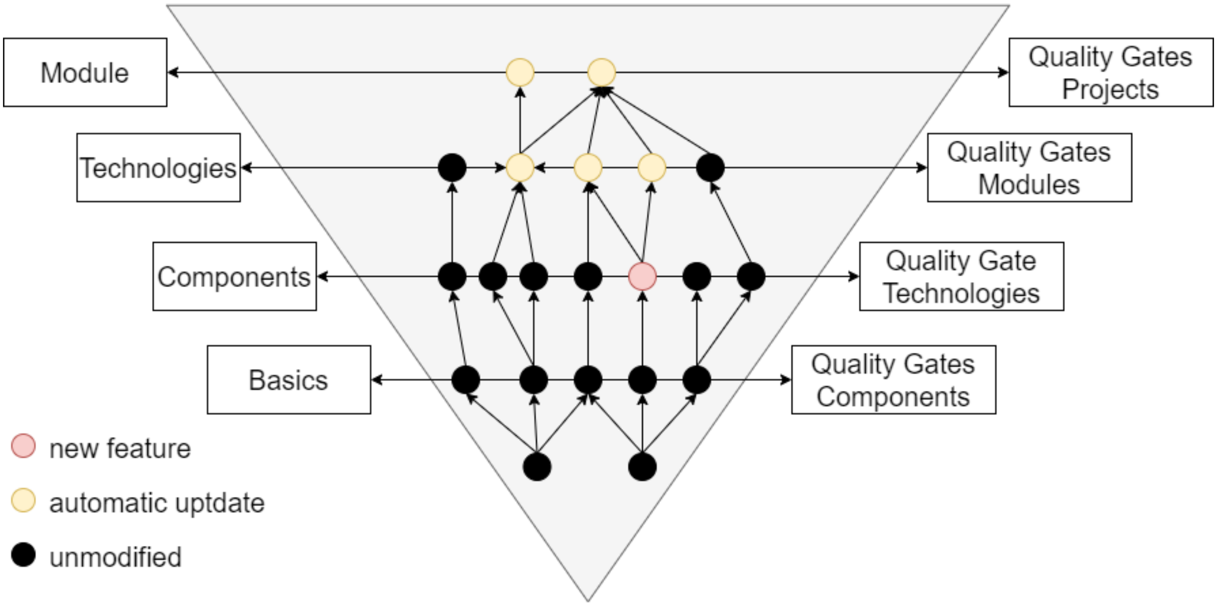


Figure 4.2: agile V-Modell of 'DigitalEngineering' department

Evaluation of the agile V-Model

According to Boehm (1999) the reuse of software components is the most effective way to reduce costs and work. (Compare [41]) In order to have a meaningful and applied reusability practice, Griss et al. (1995) suggest a policy for reusability. (Compare [42])

Problems with the internal "building block system" in Digital Engineering are finding the right modules or the knowledge about the existence of modules. If there are too many modules, the overview can be lost and too much complexity can result. Changes to a module can lead to problems if it is too integrated. Systematic approaches for solving / minimising the problems are:

- project-related naming of the repositories
- module versioning
- Dependency management (dependency update manager)
- Component testing
- documentation
- uniform code quality via style guides
- own management consultant

Thus, a systematic strategy inside Digital Engineering can be identified.

The study by Baldassarre et al. (2005) indicates that without systematic reusability, complexity actually increases. However, with a systematic reusability strategy, there is a significant reduction in complexity and an increase in reusability. (Compare [43])

Mohagheghi and Conradi (2007) have highlighted that smaller components are reused more often than medium or large-scaled modules. (Compare [44])

Tomer et al. (2004) have proposed the following model ([45]):

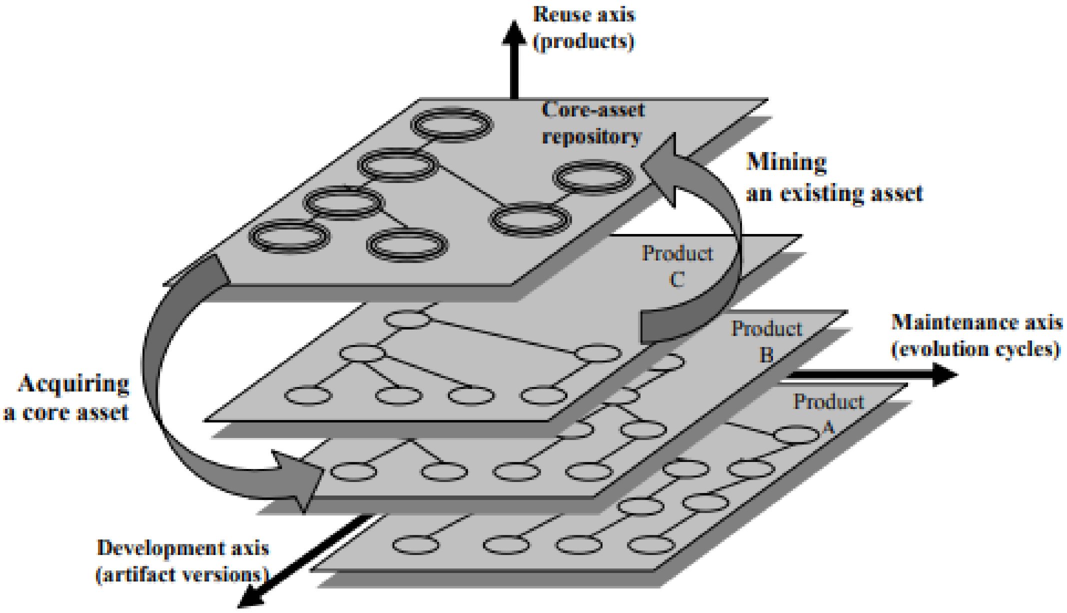


Figure 4.3: model for reuse by Tomer et al. (2004)

It has similarities to the internal model of Digital Engineering. Tomer et al. (2004) have added a third dimension, "evolution cycles", but it has, just like Digital Engineering, the range of individual artifacts in the y-direction and an increasingly compressed assembly into final products on the x-axis. The BHS model, with its strict and automatic update policy, eliminates the third axis, the "maintainance axis", since the update mechanism generally keeps the repositories up-to-date in all their dependencies.

4.2 Old Dependency-Update mechanism

4.2.1 Background of the Old Mechanism

The old update mechanism hasn't been intended for the scale it is currently used. Originally as a single developer's script, the logic has been adopted over time as a general mechanism. Within the internal model, units are kept granular and tested for integration errors or functional errors over several levels. If all test runs are successful, a safe integration of modules into other projects can be assumed. The developed dependency update mechanism threrefore only automates the corresponding triggering of the updating process of the modules.

4.2.2 Features of the old mechanism

The old mechanism first clones the inventory repository and reads all files named after the repository name. It then executes a dispatch event for each item listed. A Github dispatch event contains specific information that can be used by workflows to perform certain actions. If it is a submodule or an NPM package, specific logic is executed for it (see Figure 4.4).

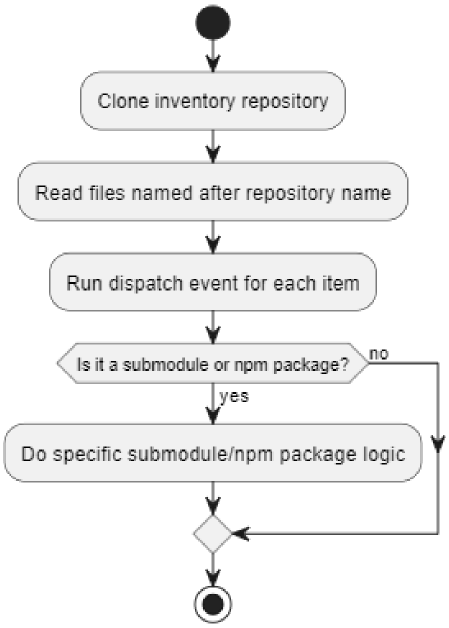

Clone inventory repository
Read files named after repository name
Run dispatch event for each item
Is it a submodule or npm package?
no
yes
Do specific submodule/npm package logic

Figure 4.4: Documentation to inventory in the old mechanism

The approach can be described as a sequential/iterative process in a simplified and more generalized form. Each repository has knowledge only about the repositories it directly depends on. After updating those repositories, the updated repositories become new starting points and proceed to update only the repositories they are aware of, see Figure 4.5.

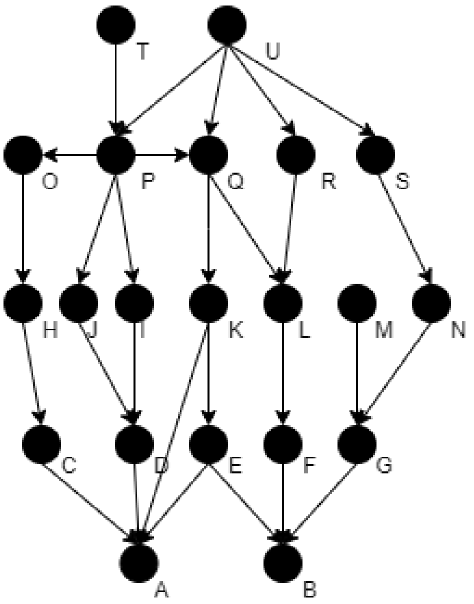


Figure 4.5: abstraction of old mechanism

4.3 Weaknesses and limitations of the old mechanism

As can be seen in Figure 4.5, repository “K” is updated two times. It will be updated because of its depedency to “A” and its dependency to “E”. Each build and update of a repository takes time and resources. Scaling up to more than 360 repositories results in unnecessary extra work.

4.4 Cost savings with improved dependency update mechanism

Duration	Current costs	30% savings	60% savings
Per hour	4,16 €	2,91 €	1,66 €
12 hours	49,92 €	34,94 €	19,97 €
4 days (96 hours)	399,36 €	279,55 €	159,74 €
30 days (1 month)	2.995,20 €	2.096,64 €	1.198,08 €

Table 4.1: Cost comparison for different savings

A single run of the dependency update mechanism takes between 12 hours (for a single trigger) and 4 days (for multiple simultaneous triggers). After 4 days, the run is generally terminated. For a minimum runtime of 12 hours, the current costs amount to 49.92 euros, while for a run of 4 days, the costs are nearly 400 euros (wasted, as the run is terminated).

According to Benedikt Bräutigram, expert software engineer at BHS, the improved dependency update mechanism is expected to reduce runner runtimes by an estimated 30% to 60%. This means that for the minimum runtime of 12 hours, there would be a reduction to 8-5 hours. If the runtime could be reduced from 12 hours to 5 hours, savings of approximately 30 euros could be achieved (49.92€ - 19.97€ = 29.95€). A 30% increase in efficiency would correspond to a reduction of about 15€ (49.92€ - 34.94€ = 14.98€). In the case of a 4-day run, the cost savings would be 240€ (with a 60% increase in efficiency) or 120€ (with a 30% improvement in efficiency).

Another important expected benefit is the reduction of the termination rate due to shorter runtimes, as the mechanism is terminated by default after 4 days of runtime. Currently, since the agile V-model is already being applied in several departments, up to 10 runners are required simultaneously during peak times (when not in use, runners are shut down and do not incur any costs).

4.5 Criticism of the old mechanism

The update process through the old mechanism is time-consuming, as it has to search and update each repository individually. All occurrences are processed one after the other without keeping an eye on the overall structure. This results in multiple accesses to the same packages and repositories.

While it is possible to trace which repository triggers other repositories for an update, it is difficult to control what common ground exists, why the reference exists. It is not possible to see whether the repositories are connected because of a submodule or, for example, because of a single NPM package. This may trigger update processes that would not have been necessary.

4.6 Benefits due to the mechanism

Despite the limitations and problems within the mechanism, 15,673 commits could be started automatically during the period April 2022, to April, 2023. Within the period, there have been only two problems due to updates of modules in projects. The mechanism thus fulfils its purpose, only the efficiency needs to be improved.

Chapter 5

Literature overview and state of the art

The approach of the Digital Engineering program of the agile V-model is not a standardized solution. Therefore, the current state of the art for managing dependencies will be examined in the following and compared with the requirements of the BHS work environment.

Updating packages and their dependencies is a necessary task in software development to ensure the security, stability, and compatibility of an application. Automated solutions such as DevBots, as presented in the studies by Erlenhov (2021 and 2022) ([14], [15]), can help to facilitate and accelerate this process. The study by Wessel et al. (2018) stated that in 2017, 26% of the open source projects they studied already included bots for support, and the trend was rising (see [17]). A possible alternative to implementing a self-developed dependency update mechanism with GitHub Actions is to use DevBots.

5.1 DevBots

DevBots are a new type of software tools designed specifically for programmers to support their daily work. (Compare [14], [15]) DevBots work automatically and autonomously and can perform tasks such as managing dependencies, updating libraries, checking code quality, and running tests. DevBots are often referred to as "artificial colleagues" or "virtual assistants" and can help developers to work more efficiently and save time.

Erlenov's articles note that there are several types of DevBots, including build bots, code analysis bots, and dependency management bots such as Dependabot, Renovate, and Greenkeeper. All three tools focus on dependency management in software projects and are designed to help developers by automating update processes.

5.1.1 Dependabot

Dependabot is a bot that automatically creates pull requests for developers to update dependencies in repositories. The bot uses open source databases such as Rubygems or NPM to gather information about available updates and then automatically suggests changes to the dependency file of the repository.

5.1.2 Renovate

Renovate is a bot that automatically updates dependencies in repositories by creating pull requests that suggest changes to dependency files. bot can be configured flexibly and supports a variety of package managers and programming languages.

5.1.3 Greenkeeper

Greenkeeper is a bot that focuses on updating NPM dependencies. The bot tracks changes in dependency files and automatically creates pull requests when an update is available. A special feature of Greenkeeper is that the bot uses a sandbox environment to run tests on the updated repository before merging changes into the main branch.

5.1.4 Further Alternatives

Snyk-Bot

Snyk is a provider of tools for security monitoring of dependencies in software projects. The Snyk Bot assists in monitoring and updating dependencies and is described by Snyk as an"automatic security and licensing bot". The article [14] does not explicitly classify Snyk Bot as a dependency management devbot. However, it mentions that Snyk is a platform for open source dependency management and that Snyk Bot is a dependency vulnerability detection tool. In addition, the Snyk company bought Greenkeeper in May 2020. Among other things, Snyk offers permanent scanning of packages and their assessment and evaluation for vulnerabilities with the help of its own "Vulnerability Database".

Pyup.io

Pyup.io is a dependency management DevBot. Pyup.io monitors dependencies of Python packages and automatically updates them when new versions are available. Pyup.io can automatically create pull requests to deploy dependency updates and run tests to ensure that the updates do not cause problems. Pyup.io also provides a security scanner that detects known vulnerabilities in Python package dependencies and notifies users when updates are available.

5.2 Limitations and problems

He et al. (2022) note some practices for avoiding or resolving update problems: "However, all these packages are mainly used as devDependencies (static typing, linters, and module bundlers), which are typically only used for development but not in production. Since the necessity of updating devDependencies remains controversial [21], [23], Dependabot's frequent and massive updates to devDependencies may be the first alarming signal of causing noise and notification fatigue to developers." ([16])

The study by Erlenhev et al. (2021) [15] shows that updating packages and dependencies regularly is a necessary but also time-consuming task for developers. DevBots such as Dependabot, Renovate, or Greenkeeper can remedy this by creating automated pull requests for updated packages. However, the study by He et al. (2022) points out that Dependabot "is effective in notifying developers to update dependencies, but has several limitations in automating dependency updates and can hardly reduce dependency management workload for developers." ([16])
One possible reason for developers' reluctance to update dependencies could be the added responsibility and effort, as He et al. also points out:
"Nonetheless, it requires not only substantial effort but also extra responsibility from developers. Consequently, there is no surprise that many developers adhere to the practice of 'if it ain't broke, don't fix it' and the majority of existing software systems still use outdated dependencies." [16]

In addition, the frequent and massive updating of devDependencies by Dependabot can lead to notification overload and fatigue for developers, as the need to update devDependencies is controversial. (Compare [21] [23])

He et al. (2017), citing the study by Mirhosseini and Parnin, state that "only 32% of Greenkeeper PRs are merged because developers become suspicious of whether a bot update will break their code due to incompatibilities (i.e., update suspicion) and feel annoyed about the large number of bot PRs (i.e., notification fatigue). Since then, many similar bots have emerged, evolved, and gained high popularity, even knocking Greenkeeper out of competition. However, it remains unknown to what extent these new bots can overcome the two limitations of Greenkeeper identified by Mirhosseini and Parnin [24] in 2017." ([16])

Another limitation of Greenkeeper mentioned in the study by Rombaut et al. (2023) is that automatically resetting broken updates to an older version is not an effective solution. (See [25])

"Reverting a broken dependency update to an older version, which is a potential solution that requires the least overhead and is automatically attempted by Greenkeeper, turns out to not be an effective mechanism. Finally, we observe that 56% of the commits referenced by Greenkeeper issue reports only change the client's dependency specification file to resolve the issue." ([25]) One reason for this is the disclaimer. If problems arise due to automated and unauthorised operations, the companies offering the algorithm could be legally vulnerable.

Another point is that DevBots and Dependency Manager (like Maven or NuGet) are specialised for a limited amount of package types and languages. The goal of the dependency update mechanism for the future is universal updating, regardless of the programming language. For the time of writing, the mechanism is intended to be applicable to NPM and NuGet packages and Submodules, but an extension for other package sources is to be made.

5.2.1 Companies experiences with Dependabot

The Digital Engineering program has already used Dependabot for testing purposes. GitHub is an online platform for version management of software projects. GitHub offers features for hosting code repositories, managing code permissions, and collaborating on projects. The platform is available for open source projects and for private projects.

Errors in GHES

In Contrast to the public accessible github.com, GitHub Enterprise Server (GHES) is a self-hosted version of GitHub. GHES allows organizations to deploy and manage GitHub-like functionality on their own servers to increase security, compliance, and control over their Git repositories. Unlike gitHub.com, where GitHub is responsible for infrastructure and maintenance, GHES is an on-premises installation run by the enterprises themselves (due to DSGVO restrictions, runners have to be hosted in Europe). Differences in the operation of Dependabot have been observed when it has been used to update dependencies in repositories on github.com and GitHub Enterprise Server. In particular, an increased error rate has been observed when using GHES. The difference may lie in the different processing and operation of Dependabot on repositories of github.com compared to GHES. During the introduction of the GitHub Enterprise Server, DependaBot has only been usable for publicly accessible repositories. It was not until 2022 (June) that an update to GHES has been introduced. This made Dependabot available for GHES as well. Nevertheless, the scope is still not the same. (Compare [28])

Costs

Furthermore, parts of the programs are subject to a fee (see [32], [35] for example). Restricting the use to individual programmers based on licenses may limit the scope of use. Extending rights to all developers, on the other hand, may exceed the costs incurred by the existing dependency update mechanism. This does not match with the goal of lowering the cost of the dependency mechanism. The offer of "JFROG Artifactory" is a minimum of USD 60,000 for the scope inside BHS. An evaluation has concluded that using Sonartype's offer for Sonartype Nexus would have cost 85,000 euros per year. Getting the dependency update function is always in a function package. It is not possible to separate out the individual functionality.

5.3 Advantages of external bots

Despite all the problems Mirhosseini and Parnin pointed out in their study about Greenkeeper, it has still found that “on average, projects that used pull request notifications upgraded 1.6x as often as projects that did not use any tools.” [24] This implies that even dependency management programs to which developers attribute poor usability to have a production increase, result in an improved time-to-market.

The management systems are also widely used and each of them has a code community forum to share ideas. (compare [29], [30], [31]) In addition, the external systems offer functionalities that can only be replicated with great effort. For example, Snyk offers its own Vulnerability Database, which can promptly draw attention to problems with packages (see [33]).

Another example is the Code Checker, which checks for common security risks and critical errors during the upload process and can issue warnings. (Compare [34])Many of these management bots or systems can be used free of charge to a certain extent.

5.4 Conclusion

While the presented DevBots offer valuable functionalities like the Code Checker, they do not fulfill all the requirements of the company. To include in-house packages, these tools need to be connected to a local dependency manager such as JFrog Artifactory or Nexus. However, it is unclear if such an integration would meet all the company’s requirements for the dependency update mechanism. Moreover, there are additional costs and efforts involved in integrating and training local dependency managers. Using different tools for different package types would increase the effort required to manage dependencies from outside.

One crucial factor that external DevBots do not cover is the dependency management of the submodules of Github. Public available Github Actions can update submodules but are hard to integrate into the company’s workflow (see [36]) or are no longer maintained (compare with [37]).

Chapter 6

Methodology

6.1 Purpose

The objective of the dependency update mechanism is to identify and keep NPM packages, NuGet packages, and submodules up to date while making their information easily accessible in a structured format. To maintain the efficiency of the format, only absolutely necessary information are stored, which represent the smallest common denominator of required information for unique identification, usage location, and corresponding version. The mechanism should be open for extensions of any package and dependency manager, for example, Maven packages, Docker images or Gradle packages.

German et al. (2007) describe the information requirements for managing inter-dependencies between modules. In this paper, however, the scope of the required information has been reduced and adapted. Information about dependencies is determined dynamically based on the algorithm, not statically fixed.

Similar to German et al, a graph algorithm / graph representation is used to illustrate and resolve dependencies. To improve efficiency, a recursive depth-first search with subsequent topological sorting has been implemented as the algorithm. This chapter provides a detailed explanation of the fundamental concepts and functionalities.

6.2 Background information

In the following section, all relevant components for the concept of new update mechanism are explained.

6.2.1 NPM

The Node Package Manager (NPM) allows developers to create, publish, and install reusable Node.js code modules (packages). It offers the possibilityof quickly installing and managing third-party packages written in Javascript via the command line. Another way to search, find, and install NPM packages is the NPM registry. “The registry is a large public database of JavaScript software and the meta-information surrounding it.” ([53]) NPM supports semantic versioning, which helps ensure compatibility between packages. All packages used by a developer are defined in the so-called “package.json” file.

6.2.2 NuGet

“Nuget is a free, open source, package management tool for the .NET platform, C++ and JavaScript. It was developed by Microsoft [...]. NuGet enables .NET developers to easily find packages, including any dependencies, and manage them.” (p. 13 [54]) Both open source and enterprise versions are available for NuGet packages. Packages are versioned using a semantic versioning scheme, which allows easy tracking of changes and updates. Comparable to NPM registry, all NuGet packages are hosted in the NuGet Gallery.

6.2.3 Submodules

Python uses pip packages (PyPI), .NET uses NuGet packages, and JavaScript provides NPM packages. However, other high-level languages do not necessarily have a native package management system, for example C or C++. With "Git submodules" this can be overcome and code fragments can be exchanged in a similar way.

It is not advisable to manually embed a Git repository within another Git repository, as this can lead to conflicts and version control management issues. Manually embedded repositories usually remain untouched if updates are done.

Submodules, however, provide a way to manage Git repositories within another repository without having to embed them manually. They offer benefits such as clean separation of repository content, easy dependency management, and easy integration of changes from external repositories. Updating submodules and setting them to specific versions is a straightforward way to ensure proper functionality of all dependencies. Thus, all the necessary functionality for the dependency mechanism is provided with submodules as well.

An important step is to have a main project that always points to the latest commit SHA of a submodule. If this is not kept up to date, new clones of a repository point to an outdated commit. If updates are executed from the clone in the submodule, then these updates start from the outdated SHA state.

6.3 JSON format for relevant information

The relevant information of the packages and submodules will be recorded in a structured “JavaScript Object Notation” (JSON) file. Figure 7.3 represents the structure.

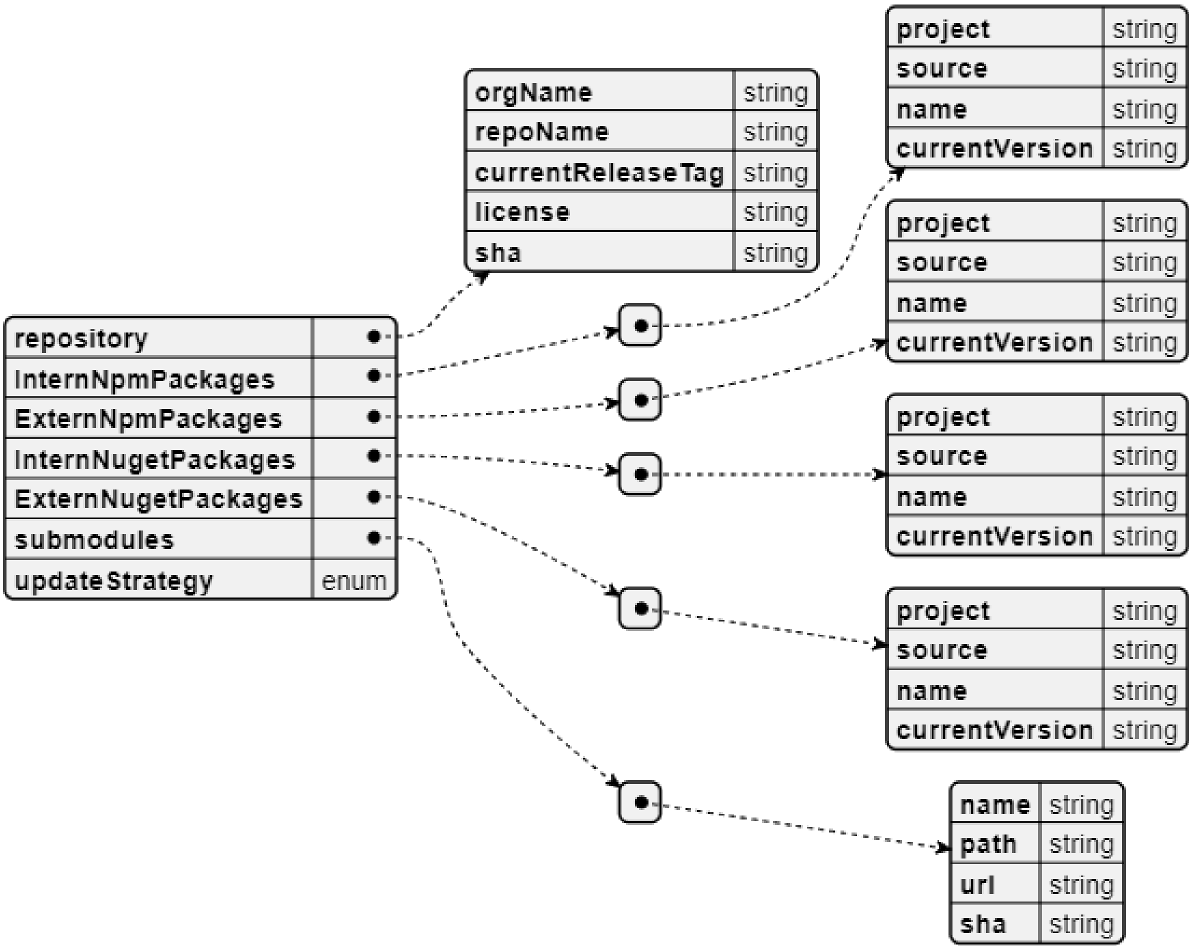


Figure 6.1: General JSON structure

JSON is a very lightweight and easy-to-read data format compared to XML and databases. It is very efficient in the use of storage space and bandwidth since it is a text based format and without the need of a complex structure. It is also very easy to parse and generate, making it ideal for use in programming languages and APIs. Compared to databases, JSON offers the advantage of easy portability and flexibility. It is very easy to transfer JSON data between different systems and applications without requiring complex database infrastructure. In addition, JSON does not require any special management or administration like a database. (Compare [48]) The final format structure has been determined by the contractor.

There is a corresponding division into "internal" and "external" origins. Only internal packages and submodules can be considered when assessing their impact on the functioning within the company. For NPM and NuGet packages, the same information is collected. This is possible because they can provide the necessary information. Submodules, for example, do not provide a versioning string according to the scheme of semantic versioning.

Initially, basic information about the repository is collected:

- **orgName:** Name of the organization. In this case, each department has its own organization name. In the event of an expansion of the agile V-model and networking with other departments, this information is needed for unique naming.
- **repoName:** The name of the repository is stored.
- **currentReleaseTag:** The current release tag of the repository is documented. This allows for tracing the currently working version.
- **licence:** The license is required to define under what conditions other individuals may use, copy, or modify the repository.
- **sha:** To ensure unique identification, the SHA value of the repository is also stored.

For the transitive packages (NPM and NuGet and in future Maven and other), the following information are required:

- **project:** NPM and NuGet packages can be used in multiple separate projects within a repository.
- **source:** The associated source is needed to determine the origin (internal or external).
- **name:** The name of the package is required for unique identification.
- **currentVersion:** The version of the package currently used is recorded.

For submodules, different information is available:

- **name:** The name of the submodule is collected for identification.
- **path:** The path is needed to specify the directory where the submodule is installed in the main repository and can be found within the repository.
- **url:** The URL is needed for unique identification
- **sha:** The SHA can theoretically be used for unique identification and as a replacement for the version number. In this case, since the name is already recorded (and each original repository is assigned a unique name), the SHA value serves as a replacement for the versioning string, as each commit has a unique identification number.

6.4 Directed acyclic graph

For the representation and processing of the dependencies, a recursive depth-first search with topological sorting is used. This requires an acyclic graph. An ordered update chronology is to be realized.

6.4.1 Definition

A graph G is a tuple *(V, E)*, where V is a set of nodes, and E is the set of edges. Multiple edges are not allowed, so only a subset of the Cartesian product $V \times E$ is formed. The definition of directed graphs is given by a pair *G =(V, E)*, where

- V is a finite set of nodes, and
- E is a set of edges.

However, each e ∈ E is now a tuple (a, b) with a, b ∈ V. "This definition, however, allows for cyclic graphs." [38] If a directed graph contains cycles, that is, the possibility to reach the same point again in the direction of an arrow (on a round trip), then it is called cyclic. (Compare 6.2) It represents an arbitrary mathematical relation, but not an order relation. If a directed graph does not contain cycles, it is called acyclic. Mathematically, it represents a partial order. In object-oriented programming, a polyhierarchy corresponds to a directed acyclic graph. (Compare [38])

6.4.2 Complexity

The following table lists the complexity of various operations for different representations of a graph. It holds that n = ∣V∣ and m = ∣E∣.

Operation	Edge List	Node List	Adjacency Matrix	Adjacency List
Insert Edge	*O(1)*	*O(n + m)*	*O(1)*	*O(1) / O(n)*
Delete Edge	*O(m)*	*O(n + m)*	*O(1)*	*O(1)*
Insert Node	*O(1)*	*O(1)*	*O(nˆ2)*	*O(1)*
Delete Node	*O(m)*	*O(n + m)*	*O(nˆ2)*	*O(n + m)*

Table 6.1: Complexity of Graph perations ([38])

In general, it can be stated that an adjacency list is used because it is estimated to be the more efficient solution in the course of the application on a test repository. If the concept of the dependency update mechanism is expanded, the efficiency may decrease in favor of an adjacency matrix.

While the mechanism is said to be designed for efficiency, it is not located in a time-critical environment. Therefore, the exact specification of the complexity can be neglected.

6.4.3 Depth-first Search

"A depth-first search of a graph $G = (V, E)$ systematically explores each vertex and each edge in the graph. The algorithm proceeds as follows. First, each vertex is marked as unexplored by coloring it white. Then the search begins. The search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. If all of v's edges have been explored, the search 'backs up' to explore edges leaving the vertex from which v has been discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex." ([40])

Algorithm 1 DFS(G)

```
Require: Graph G
for each Node $u \in V(G)$ do
    $color[u] \leftarrow white$; $\pi[u] \leftarrow null$
end for
$time \leftarrow 0$
for each Node $u \in V(G)$ do
    if $color[u] = white$ then DFS-visit(u) fi
end for
```

Algorithm 2 DFS-visit(u)

```
Require: Node u
for each $v \in$ Adj(u) do
    if $color[v] = white$ then
        $\pi[v] \leftarrow u$; DFS-visit(v)
    fi
end for
$color[u] \leftarrow black$; $time \leftarrow time + 1$; $f[u] \leftarrow time$
```

([38])

6.4.4 Breath-first search

Given a graph G D .V; E/ and a distinguished source vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. [**?**] p. 594

6.4.5 Cycle detection

"A directed graph is acyclic if and only if a depth-first search of the graph yields no back edges. In contrast, if a depth-first search does have a back edge (u, v), then vertex v is an ancestor of vertex u in the depth-first tree, which implies that there is a path from v to u in the original graph that starts with a back edge. [...] To keep track of

Algorithm 3 BFS(G, s)

```
Require: Graph $G$, a start node $s \in V[G]$
for each Node $u \in V(G) - s$ do
    $color[u] \leftarrow white; d[u] \leftarrow \infty; \pi[u] \leftarrow null$
end for
$color[s] \leftarrow grey; d[s] \leftarrow null; \pi[s] \leftarrow null$
$Q \leftarrow emptyQueue; Q \leftarrow enqueue(Q, s);$
while $\neg isEmpty(Q)$ do
    $u \leftarrow front(Q);$
    for each $v \in DestinationNodeOutgoingEdges(u)$ do
        if $color(v) = white$ then
            $color[v] \leftarrow grey; d[v] \leftarrow d[u] + 1;$
            $\pi \leftarrow u; Q \leftarrow enqueue(Q, v)$
        fi
    od
    $dequeue(Q); color[u] \leftarrow black$
od
```

progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is discovered the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore,´have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner" ([47]). So a white colored node are unmodified. If a node is marked as grey, they already have appeared at least once and the distance has been determined. But unlike black nodes, no distance has yet been calculated from them.

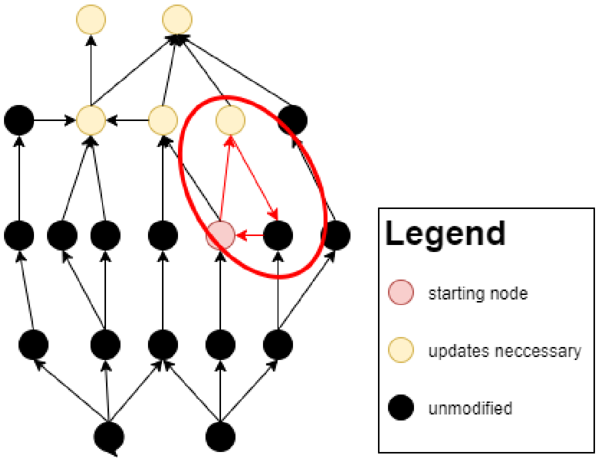


Figure 6.2: Cycle inside graph

6.4.6 Topological Sorting

"A topological sort of a dag *G = (V, E)* is a linear ordering of all its vertices such that if *G* contains an edge *(u, v)* , then *u* appears before *v* in the ordering." ([46])

Algorithm 4 TOPOLOGICAL-SORT(G)

```
Require: Directed graph G
Call DFS(G) to compute finishing times $v.f$ for each vertex $v$
As each vertex is finished, insert it onto a linked list
Return the linked list of vertices
```

6.5 Conclusion

The basic theory for the design of the new dependency update mechanism has been explained. First, it has been described which information has to be prepared in what way. Then the origin of the algorithm for the recursive depth-first search with topological sorting has been demonstrated. This serves to understand the concrete implementation and the motivations for individual decisions.

From the theory, two GitHub Actions have been designed that put the requirements into reality. This is described in the following chapter.

Chapter 7

Implementation

7.1 Description of the implemented solution

This implementation is part of an automated dependency update mechanism that allows developers to keep certain packages and modules of their projects up to date by automatically updating outdated packages / modules.
Two separate Github Actions have been implemented:

1. "Write2Inventory": The task of the first action is to collect and archive the desired information in a uniform JSON format.
2. "DependencyUpdate": The second action uses the previously generated JSON files to create a graph and a topological sorting of the update order of all relevant repositories.

GitHub Actions, an automation system, is a CI/CD solution integrated into GitHub. Actions can be used to start and control automated software processes such as bug management, project management, or code safety analysis. (Compare [49])

7.2 Action No. 1: Write2Inventory

7.2.1 Purpose of the Action

The first action, Write2Inventory, collects basic information for each repository that is to be considered, as shown in Figure 7.3. The code implements an algorithm to gather information about NPM and NuGet packages as well as submodules for a specific GitHub repository.

7.2.2 Code and functionality

The final output object has a defined structure and contains information about the repository, internal and external NPM and NuGet packages, as well as internal and external submodules. In addition, the strategy for updating packages is determined. Finally, the output object is formatted as a JavaScript Object Notation (JSON) string and then exported to a separate repository.

Collecting repository information

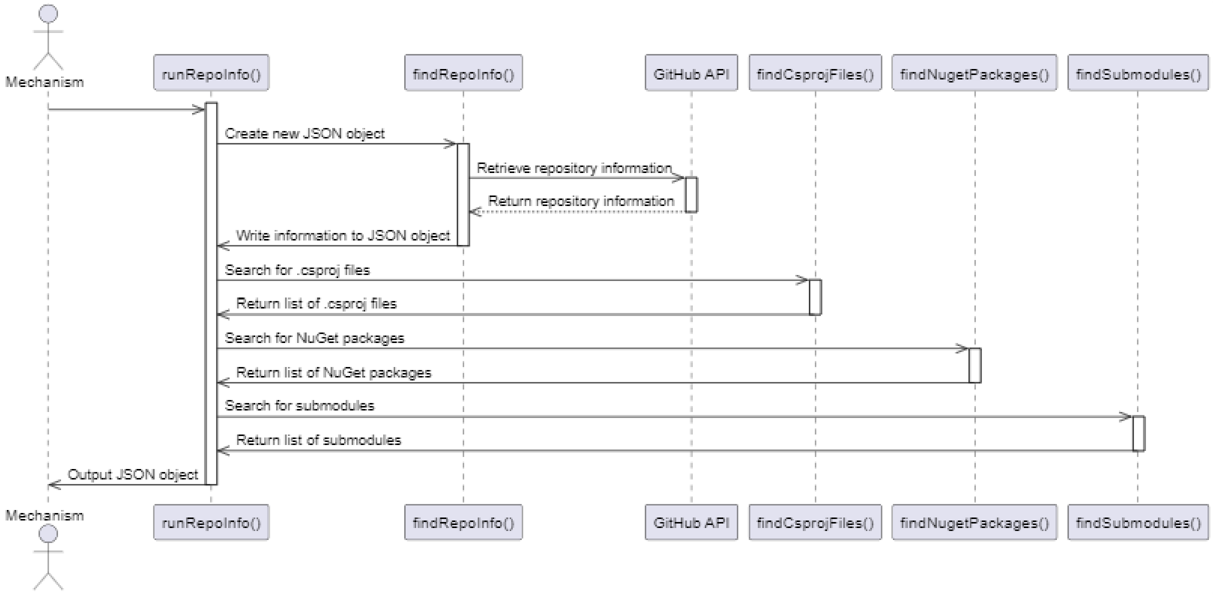


Figure 7.1: Sequence diagram of the central method

The following section will explain in detail the relevant methods (compare 7.1) for gathering information:

findALLCSPROJmodules()

The purpose of this method is to find all files in a repository that end with “`.csproj`”, including submodules. The method calls two git commands:

1. ‘‘`git submodule update -init -recursive`’’: The main part of the command is ‘‘`git submodule update`’’. This section updates all submodules to the state of

the current commit in the repository where it is included as a submodule. This repository may be different from the original repository where the submodule has been created (see Figure 7.2).

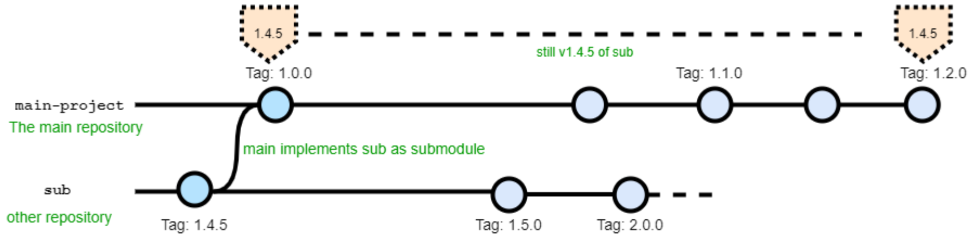


Figure 7.2: Integration submodule

The option "-init" is required to initialize the submodules. The entire method is embedded in the GitHub Action Write2Inventory", which temporarily clones the respective repository where it is applied. If the option `-init` is missing, the subdirectory for the submodule would be empty, requiring a separate initialization of the submodules. With the option "`-recursive`", submodules of submodules are also recursively considered and included in the process of the command.

2. "`find . -name .csproj`": This command finds and lists all files with the "`.csproj`" extension in a repository, including in submodules and subdirectories.

.

The return value of the method is ultimately a flat list of paths (relative to the working directory where the code was executed) to the .csproj files found.

findNugetPackages()

The purpose of this method is to determine a list of all NuGet package sources used.

The command "`dotnet nuget list`", called by the method, lists all configured NuGet package sources in the directory. The option `-format short` reduces the output of the command to the URL of the package sources. Additionally, it shows whether the package source is enabled or disabled. Disabling a source means that NuGet will ignore it and will not search for or download packages from it. The method returns a list of URLs used as NuGet package sources for the .NET application.

getAllNuGetPackages()

The output of the two methods `findALLCSPROJmodules` and `findNugetPackages` are used as input parameters in the method `getAllNuGetPackages`. The aim of the method is to output a list of all NuGet packages considered as internal ("`https://nuget.github.bhs-world.com/...`") or external (for example "`https://api.nuget.org/v3/index.js`").

The method calls the .NET command ‘‘`dotnet list $project package -source $source`’’ with the ‘‘`project`’’ and ‘‘`source`’’ parameters provided iteratively by using the lists obtained from `findALLCSPROJmodules` and `findNugetPackages`. The output objects contain:

- `project` (the name of the project that contains the package)
- `source` (the URL of the source from which the package was installed)
- `name` (the name of the package)
- `currentVersion` (the current version of the package)

of each package, which is then separated into internal and external package objects based on a predefined list of sources (see above). If a package comes from a source declared as internal, it is added to the list of internal packages, otherwise it is defined as external package.

getAllNPMPackageInfo()

The purpose of the method is to collect desired attributes such as the name of the project it is used in, the name of the package, URL source, and the currently used version of the NPM package across all NPM packages within a repository. The package objects are then divided into external and internal package objects based on a list of valid source URLs (starting with ‘‘`@digitalengineering`’’, ‘‘`@requipment`’’, . . .). Internal means that the NPM package originates from the GHES itself. External packages, on the other hand, are public and freely available packages (for example ‘‘`@actions/core`’’, ‘‘`@actions/github`’’, `...`).

First, the method calls the NPM command `npm ls -depth=0 -json` and extracts the name of each package from its output, saving them in a list. Then, for each package in the list, the method calls the `getSingleNPMPackageInfo` function to obtain more detailed information about each package. The function returns an object of type `Packages`, which includes the following fields:

- `project` (the name of the project that contains the package)
- `source` (the URL of the source from which the package was installed)
- `name` (the name of the package)
- `currentVersion` (the current version of the package)

.

Next, the `Packages` are divided into internal and external packages. The criterion for separation is a list of valid source references (see above). If the source of a package is found in the list of internal sources, the corresponding package is considered internal, otherwise it is considered an external NPM package.

getSubmodules()

The method `getSubmodules()` is used to read all submodules of a repository. Based on a list of internal submodule URLs (for example ‘‘`https://github.bhs-world.com/...`’’), a distinction is made again between submodules of internal or external origin. Internal submodules come from the company’s GitHub Enterprise Server, while external submodules correspond to modules of external origin, such as from a publicly accessible repository.

The method first calls the git command `git submodule status -recursive`. The command returns a list of all submodules of a repository. The `-recursive` option also lists submodules in submodules recursively. Then, the desired data are extracted based on the `Submodule` interface, which includes the following properties:

- `name` (the name of the project that contains the submodule)
- `path` (relative path of the submodule in the repository)
- `url` (full URL of the repository, along with the source information)
- `sha` (current unique SHA of the submodule)

.

Afterwards, the submodules that were found are divided into internal and external submodule objects according to the list of internal submodules and are returned.

7.2.3 Conclusion

The collected data are stored in a separate JSON file for each repository in a central GitHub repository (called "Inventory") (see Figure 7.3). Each file is named after the repository it refers to. In the end the file structure gets filled with basic repository information, information about each package and each submodule inside the repository.

An example output is provided in the Appendix A.3. The files build the basis for the subsequent GitHub Action "DependencyUpdate".

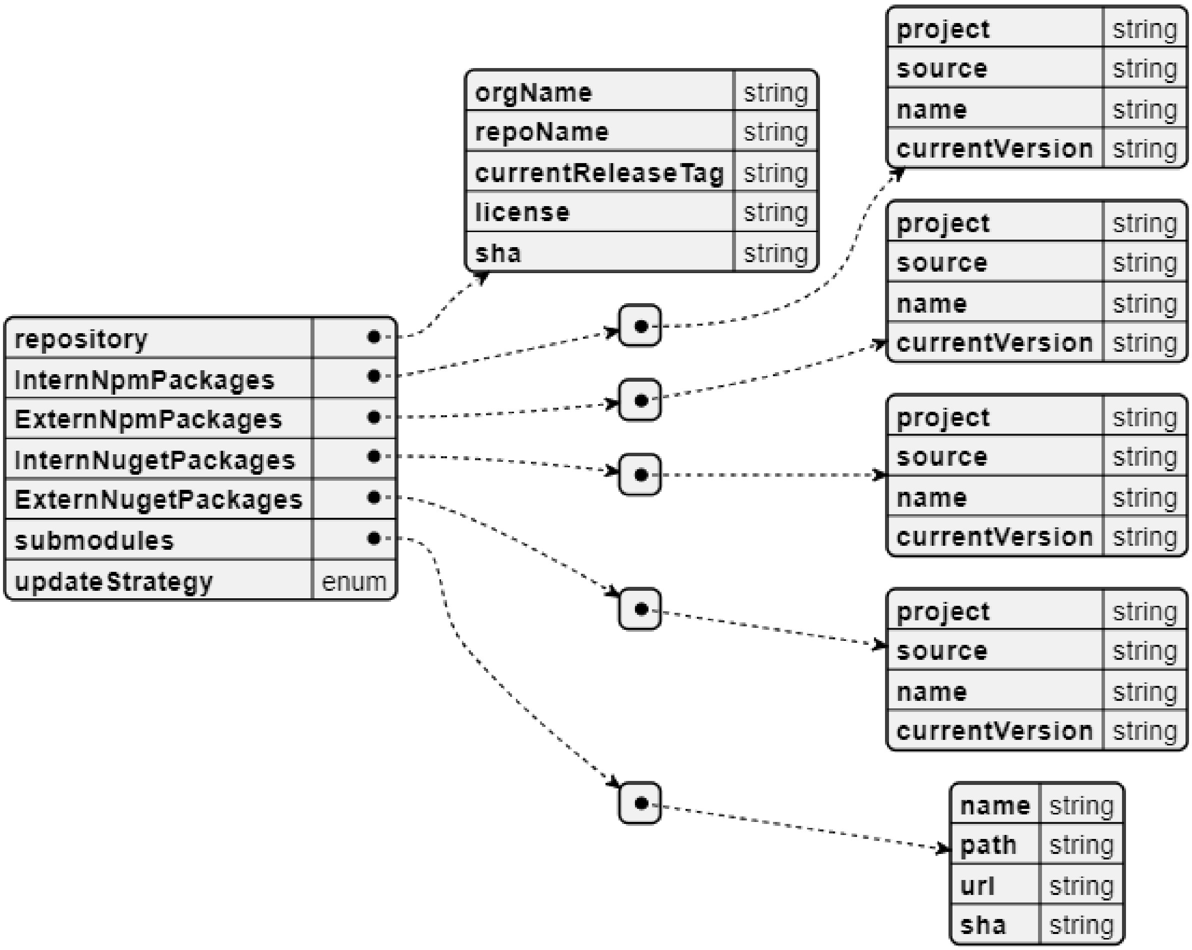


Figure 7.3: general json structure

7.3 Action No. 2: "DependencyUpdate"

7.3.1 Purpose of the Action

This GitHub Action reads all JSON files written to the Inventory and collects the information to form a graph. By resolving the dependencies using a recursive depth-first search and topological sorting, it creates an order in which the repositories should be updated without triggering a conflict in the order.

7.3.2 Code and functionality

This action has two main tasks: After reading all JSON data from the central "Inventory" repository, first, the respective dependencies are created, and then, they are evaluated, resolved, and finally put in order based on a directed acyclic graph.

Collecting the dependencies

To create the dependencies, an abstract class `DependencyType` has been defined. `NpmDependency`, `NugetDependency` and `SubmoduleDependency` inherit from this abstract class and implement their specific `getDependencies` methods to extract the dependencies from the JSON files. (See Figure 7.4)

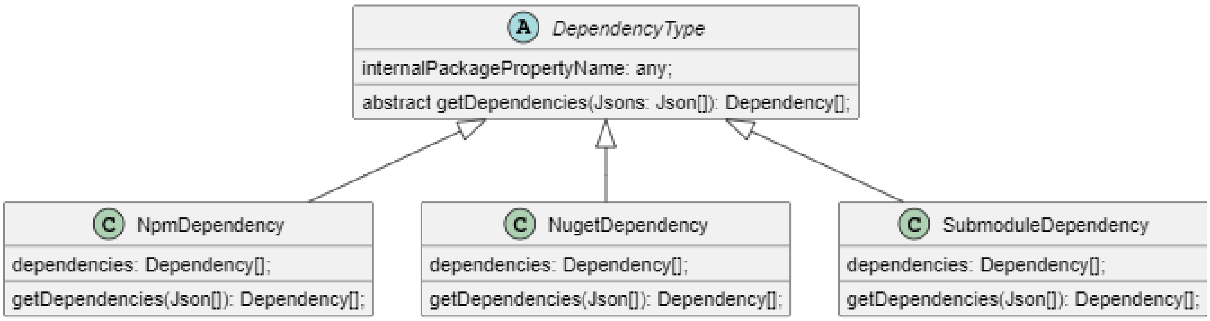

A DependencyType
internalPackagePropertyName: any;
abstract getDependencies(Jsons: Json[]): Dependency[];
C NpmDependency
dependencies: Dependency[];
getDependencies(Json[]): Dependency[];
C NugetDependency
dependencies: Dependency[];
getDependencies(Json[]): Dependency[];
C SubmoduleDependency
dependencies: Dependency[];
getDependencies(Json[]): Dependency[];

Figure 7.4: General dependency type class

The class `DependencyCollector` iterates over all `getDependencies`. The method `getDependencies()` of the `DependencyCollector` class creates a list of `DependencyType` objects. These objects contain `NpmDependency`, `NugetDependency`, and `SubmoduleDependency`. Then, the method calls the `collectDependencies()` method and returns the result.

The three classes are instantiated in `DependencyCollector` and called by the `collectDependencies()` method to finally return a list of dependency objects containing all collected dependencies from the JSON objects.

This structure follows the Single Responsibility Principle, the Open-Closed Principle, and the Dependency Inversion Principle of software development, coined by R. C. Martin in 2000 (see [50]).

`getDependencies()` of `DependencyCollector` now includes all dependencies. These are then transformed into a map structure using `getDependencyMap()`.

DependencyMap

A DependencyMap has been implemented to model incoming dependencies in a map structure. The DependencyMap can be defined as a function $f : A \rightarrow B$, where A is the set of keys and B is the set of lists of dependencies. Each key k_i from A is mapped to a list l_i of dependencies from B, i.e. $f(k_i) = l_i$.

The following example corresponds to the resolution of Figure 7.5 into a map:

DependencyMap: M = {
A $\rightarrow$ [C, D, E, K],
B $\rightarrow$ [E, F, G],
C $\rightarrow$ [H],
D $\rightarrow$ [I, J],
E $\rightarrow$ [K],
F $\rightarrow$ [L],
G $\rightarrow$ [M, N],
H $\rightarrow$ [O],
I $\rightarrow$ [P],
J $\rightarrow$ [P],
K $\rightarrow$ [Q],
L $\rightarrow$ [Q, R],
N $\rightarrow$ [S],
O $\rightarrow$ [P],
P $\rightarrow$ [T, U],
Q $\rightarrow$ [P, U],
R $\rightarrow$ [U],
S $\rightarrow$ [U]
}

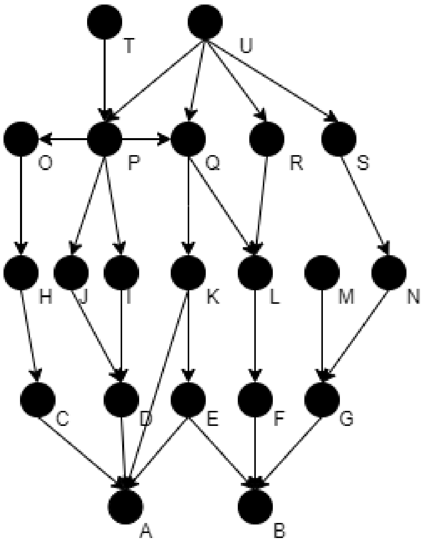


Figure 7.5: Example graph

`getDependencyMap()` iterates over each dependency in the input list and adds it to the DependencyMap. The key of the map is the name of the source repository, and the value is a list of target repositories (see Algorithm 5). Only direct dependencies are stored, no indirect/transitive dependencies. This corresponds to the recommendation of Maven.org:

“Although transitive dependencies can implicitly include desired dependencies, it is a good practice to explicitly specify the dependencies your source code uses directly. This best practice proves its value especially when the dependencies of your project change their dependencies. For example, assume that your project A specifies a dependency on another project B, and project B specifies a dependency on project C. If you are directly using components in project C, and you don’t specify project C in your project A, it may cause build failure when project B suddenly updates/removes its dependency on project C.” ([51])

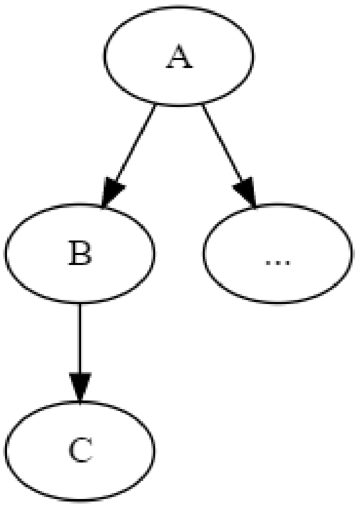


Table 7.1: Graph example for Maven citation

The operations to add an element to the map or to append to a list have a time complexity of *O(1)*. Therefore, overall, `getDependencyMap()` has a time complexity of *O(m)*, where *m* is the number of dependencies in the input list.

Algorithm 5 Determination of the DependencyMap

```
procedure getDependencyMap(dependencies)
    $map \leftarrow []$                                  ▷ Initialize empty DependencyMap
    for each $dependency \in dependencies$ do
        $from \leftarrow dependency.from$                ▷ Store source repository
        $to \leftarrow dependency.to$                    ▷ Store target repository
        if $from$ ist kein Schlüssel in $map$ then
            $map[from] \leftarrow []$        ▷ Initialize empty list for the source repository
        end if
        $map[from].push(to)$     ▷ Add target repository to the list of dependencies
    end for
    return $map$                                         ▷ Return DependencyMap
end procedure
```

reverseDependencyMap

One of the limitations of the `getDependencyMap()` function is that it only indicates which dependencies are utilized by a given object. However, the dependencies themselves are unaware of the potential impact that updates may have on the objects that utilize them. The `reverseDependencyMap()` function overcomes this limitation by indicating which objects utilize a given dependency. It thus provides a reversed view of the dependency map, converting the statement "Q utilizes K" to "K is utilized by Q". The overall complexity of this function is *O(n*m)*, where *n* is the number of modules and *m* is the average number of dependencies per module.

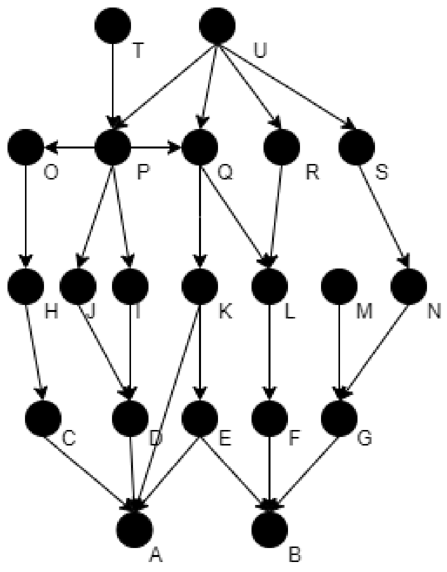


Table 7.2: Before reverseDependencyMap()

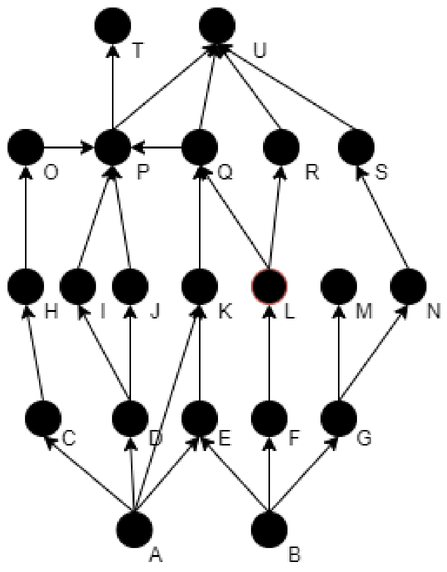


Table 7.3: After reverseDependencyMap()

topologicalSortDFSWithStart

The function `topologicalSortDFSWithStart()` takes the Map object from `reverseDependencyMap()` as input.

1. Recursive depth-first search is used to traverse the dependency graph of a package or module and collect the dependencies until all dependencies have been captured.
2. The dependency graph is sorted into a list using topological sorting, which ensures that dependencies are always updated before their dependents.

.

By performing these two steps, it is ensured that all dependencies of a package or module are updated in the correct order to avoid compatibility issues and other errors. This function has a linear time complexity of *O(n + e)*, where *n* is the number of nodes and *e* is the number of edges in the graph. In this case, *n* is the number of unique repositories in the input list contained in the DependencyMap, and *e* is the number of dependencies in the input list.
The Figure 7.4 shows the topologically sorted list. It should be mentioned that “K” occurs only once in the list. In the current mechanism, “K” and “U” would have appeared twice, “P” would have appeared three times (see chapter 4.3).

Result would be the list:
`[A,E,K,Q,D,J,I,C,H,O,P,U,T]`

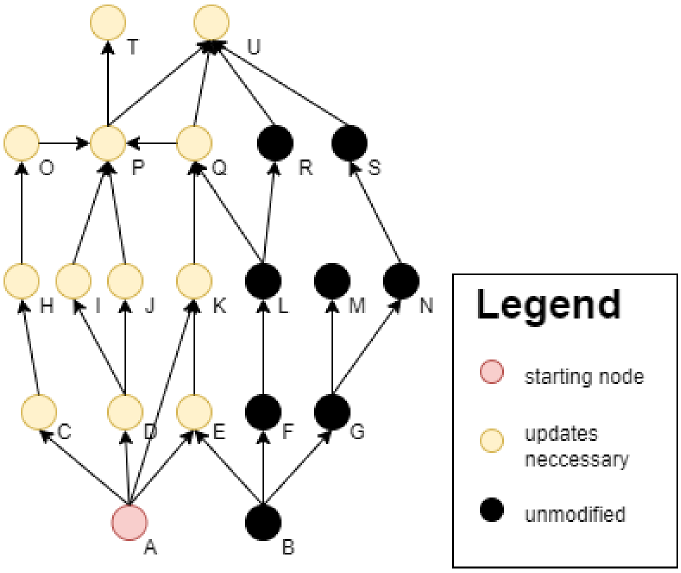


Table 7.4: Modification with chain reaction

A check for cycles has been implemented. A recursive depth-first search with topological sorting can only be applied to a directed acyclic graph. If the check detects a cycle, dependencies that trigger a cycle are resolved (see Figure 7.6). The algorithm continues, but a warning is issued to the developer that a cycle was present and that closer inspection and possibly human correction may be required.

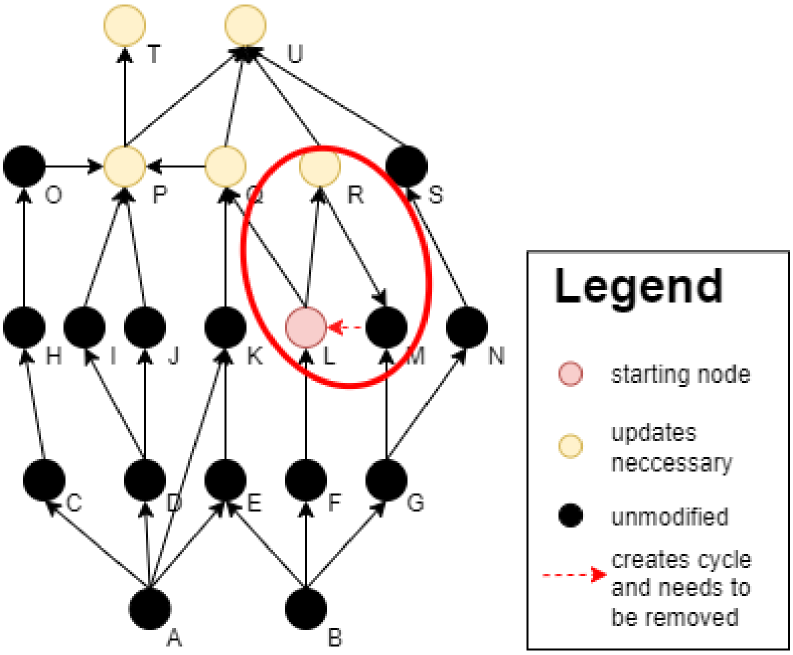


Figure 7.6: Cycle detected and resolved

Algorithm 6 Topological sorting with DFS with starting node

```
function topologicalSortDFSWithStart(dependencyMap, startNode)
    $sorted \leftarrow []$                                   ▷ Initialize empty list for sorted nodes
    $visited \leftarrow []$                                  ▷ Initialize empty set for visited nodes
    $reversedMap \leftarrow$ reverseDependencyMap$(dependencyMap)$        ▷ Reverse the dependency map
    function dfs(node)                                ▷ Perform depth-first search (DFS)
        if node is in visited then                    ▷ If node has already been visited, return
            return
        end if
        $visited[node] \leftarrow$ True                         ▷ Mark node as visited
        $dependencies \leftarrow reversedMap[node]$             ▷ Get dependencies of the node
        for each $dependency$ in $dependencies$ do
            dfs$(dependency)$                           ▷ Perform DFS on each dependency
        end for
        $sorted.push(node)$                             ▷ Add node to the sorted list
        return                                        ▷ Return sorted list
    end function
    dfs$(startNode)$                                    ▷ Perform DFS on the starting node
    return $sorted.reverse()$                          ▷ Return the sorted list in reverse order
end function
```

topologicalSortBFSWithStart

The function `topologicalSortBFSWithStart()` takes the Map object from `reverseDependencyMap()` as input.

1. Breadth-first search is used to traverse the dependency graph of a package or module and identify the dependencies until all dependencies have been captured.
2. The dependency graph is sorted into a list using topological sorting, which ensures that dependencies are always updated before their dependents.

. By performing these two steps, it is ensured that all dependencies of a package or module are updated in the correct order to avoid compatibility issues and other errors. This function has a linear time complexity of *O(n + e)*, where *n* is the number of nodes and *e* is the number of edges in the graph. In this case, *n* is the number of unique repositories in the input list contained in the DependencyMap, and *e* is the number of dependencies in the input list.

The Figure 7.5 shows the topologically sorted list. It should be mentioned that K” occurs only once in the list. In the current mechanism, K” and U” would have appeared twice, P” would have appeared four times (see chapter 4.3).

Result would be the list:
`[A,C,D,E,K,H,I,J,Q,O,P,U,T]`

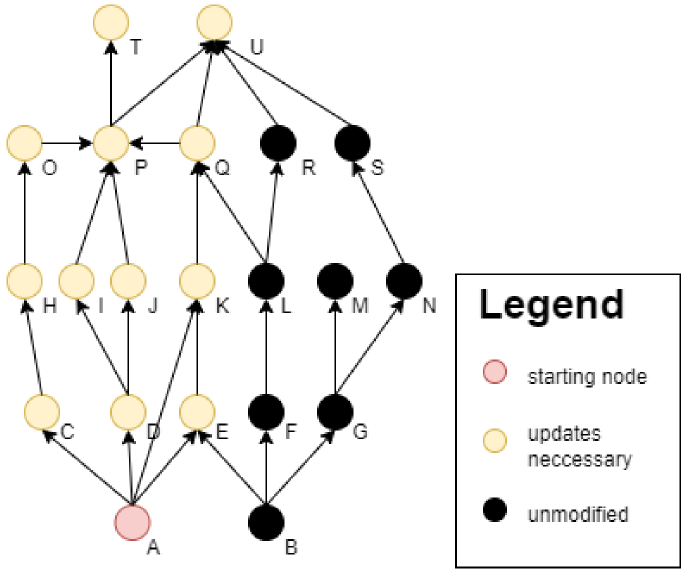


Table 7.5: Modification with chain reaction

Algorithm 7 TopologicalSortBFS(dependencyMap, startNode)

```
Require: dependencyMap: A dictionary of nodes and their dependencies; startNode:
    The starting node for sorting
Initialize an empty sorted list
Initialize an empty list of visited nodes
Initialize an empty queue
Reverse the dependencyMap to get a list of nodes that depend on it
Add the startNode to the queue
while Queue is not empty do
    Remove the first element from the queue and name it "node"
    if node has not been visited then
        Mark node as visited
        Get the dependencies of node from the reversed dependencyMap
        for each dependency do
            Add the dependency to the queue if it hasn't been visited
        end for
        Add the node to the sorted list
    end if
end while
return the sorted list
```

7.4 Challenges and limitations

7.4.1 Challenges and limitations of "Write2Inventory"

The mechanism should follow the Open-Closed Principle, which is "open for extension" ([50]). The extensibility of the JSON structure for Maven packages should be relatively unproblematic. Maven packages provide the same information as NPM and NuGet packages, so they can be added following the same pattern. However, adding Docker images may require new structures to ensure their unique identification and mapping to their deployment locations and corresponding versions.

Another limitation that needs to be carefully considered in practical implementation is the size of the repositories. Full (temporary) cloning and analysis of contents can lead to performance problems. One possible solution is to use caching to reduce the number of requests to the server.

7.4.2 Challenges and limitations of DependencyUpdate

The algorithm of recursive depth-first search with subsequent topological sorting is designed for acyclic graphs. A check on the graph is already built-in, and the resolution of cycles is also implemented. Depending on the complexity of the graph structure, additional algorithms for cycle resolution, such as Tarjan's algorithm (compare [57], [58]) can be applied. The effects of dependency resolution has to be carefully monitored by developers to ensure a smooth operation since specific edges (two or more) were deliberately removed.

Another limitation is that currently only one starting node can be specified. Extending the algorithm to multiple parallel starting positions is not yet implemented.

7.5 Conclusion

The GitHub Action DependencyUpdate uses the JSON files created by the first action, Write2Inventory. The action contains three important methods, `getDependencyMap()`, `reverseDependencyMap()` and `topologicalSortWithStart()`.

With `getDependencyMap()`, dependencies between the NPM packages, NuGet packages and submodules and the repositories in which each of them are located. `reverseDependencyMap` reverses the direction of the dependencies in a map, so that the keys become values and vice versa.

The result is picked up by `topologicalSortWithStart()`. Using the provided dependency map, a recursive depth-first search with topological sorting is applied to a corresponding graph. The result of the topological sorting is the order of the repositories for the update mechanism. The order ensures that repositories only need to be updated once during a workflow.

This automated approach saves a lot of time and effort with respect to the reference model, as multiple invocations of modules and repositories are avoided.

Chapter 8

Experimental Results

8.1 Test data and methodology

Various testing methods were applied.

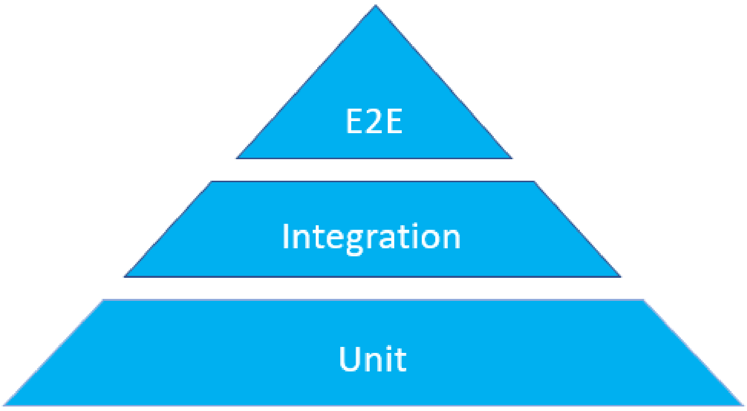


Figure 8.1: Testing pyramid

8.1.1 Test applications for Write2Inventory

In the first Action, Write2Inventory

- automated tests (unit tests)
- integration tests
- manual verification of results

were used to guarantee the smoothest possible behavior. "End-to-end tests" were not implemented. This will be part of future work.

Unit-Testing

Unit tests were created using Jest, a freely accessible test framework for JavaScript applications. Specifically, the most important functions `findCSPROJmodules()`, getDotnetSources(), getPackageInfo(), getAllPackageInfo(), and getSubmodules() have been tested. Each of the functions was first examined for its behavior when given empty or non-existent data that they require.

Then, the functions were called using test datasets, and their respective return values have been checked. The return value of each function has been compared to the expected value. Due to the rigid JSON structure that has to be adhered to, it was easy to determine whether each method passed the test or not.

Integration Testing

An integration test has been created using Jest. Unit tests check the behavior of individual components, in this case, the most important methods of the GitHub Action. The integration test aims at examining the interaction of all methods and identifying any potential errors.
The main focus of the integration test has been to verify whether the data can be correctly read and processed from a Git repository. Specifically, the assignment of internal and external NPM packages has been checked.
For the integration test, a temporary directory structure has been created to prevent changes to the existing directory structure. A repository has been cloned into this directory, and a temporary package.json file has been created, in which the relevant dependencies have been installed. Then the method getAllPackageInfo() has been applied, and it has been verified whether the allocation of the tested package was correct. In contrast to the unit tests for the method getAllPackageInfo(), its behavior was tested in interaction with real Github repositories.

8.1.2 Test applications for "DependencyUpdate"

...

8.2 Analyse und Diskussion der Ergebnisse

Chapter 9

Summary and future work

9.1 Summary of the results

A concept for improving an existing dependency update mechanism was designed and implemented. Previously, the current state of the art was addressed and no way was found to achieve the same effect. Third-party solutions that have been found are either beyond the monetary scope and / or are limited in their mode of operation, in relation to the requirements of the BHS. Initial tests indicate that an increase in efficiency, and thus an increased monetary benefit, can be achieved through the newly implemented mechanism.

The environmental structure in which the mechanism is incorporated (the internally used agile v-model) was examined and its functioning confirmed. Found models of a similar nature were found and compared with the internal model. It has been proven that the problems that occur are well handled by the systematic management and the management agenda.

The structure of the agile v model, together with the dependency update mechanism can be exported to other departments. The export of the agile V-model will with its dependency update mechanism will ensure a faster assimilation of the less digitalized departments to the highly networked and digitalized way of dealing, in line with the Industry 4.0 agenda.

9.2 Future work

In order to fully integrate the update process, it is necessary to implement a third Github Action and integrate all three actions into the overall process (see A.1). Additional test scenarios with a more complex structure can also be added to improve the safety and reliability of the function.
One potential area for future research could be exploring the optimization of the Depth-First-Search algorithm using the Depth-First Discovery Algorithm developed by Zhou and Müller (2003). This approach could potentially reduce resource and time consumption as complexity increases [59].

Additionally, it may be worth considering the integration of aspects from aspect orientation into the agile V-model. While the agile V-model incorporates concepts of object orientation, aspect orientation is considered to be an extension of this approach [60]. This could potentially lead to further improvements in the efficiency and effectiveness of the development process.

Bibliography

[1] BHS Corrugated, "BHS Corrugated: Mechanical Engineering and Digital Solutions for the Corrugated Board Industry," [Online]. Available: https://www.bhs-world.com/en/company. [Accessed: Apr. 20, 2023]

[2] BHS Corrugated, "Facts and Figures - BHS Corrugated Group," [Online]. Available: https://www.bhs-world.com/en/company/facts-and-figures. [Accessed: Apr. 20, 2023]

[3] Staufen AG, "German Industry 4.0 Index 2022," Staufen AG, 2022. [Online]. Available: https://www.staufen.ag/insights/studien-whitepaper/studie-deutscher-industrie-4-0-index-2022. [Accessed: Apr. 20, 2023]

[4] Statista, "Fachkräftemangel in MINT-Berufen," [Online]. Available: https://de.statista.com/infografik/20030/fachkraeftemangel-in-mint-berufen/. [Accessed: 22-Mar-2023].

[5] Statista, "IT-Fachkräftemangel in Unternehmen in Deutschland," [Online]. Available: https://de.statista.com/statistik/daten/studie/795219/umfrage/it-fachkraeftemangel-in-unternehmen-in-deutschland/. [Accessed: 22-Mar-2023].

[6] Y. Jiao et al., "A cloud approach to unified lifecycle data management in architecture, engineering, construction and facilities management: Integrating BIMs and SNS," Advanced Engineering Informatics, vol. 27, no. 2, pp. 173-188, 2013

[7] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in IEEE International Conference on Software Maintenance. ICSM, 2013, pp. 70-79.

[8] C. Bogart, C. Kästner, and J. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), 2015, pp. 86-89.

[9] S. Bier et al., "Mitigating remote code execution vulnerabilities: a study on tomcat and android security updates," in 2021 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS), 2021, pp. 1-6.

[10] National Vulnerability Database (NVD), [Online]. Available: https://nvd.nist.gov/general. [Accessed: 28-Feb-2023].

[11] FIRST, "CVSS v3.1 Specification Document," FIRST, 28-Feb-2023. [Online]. Available: https://www.first.org/cvss/v3.1/specification-document. [Accessed: 22-Mar-2023].

[12] NIST, "National Vulnerability Database," [Online]. Available: https://nvd.nist.gov/vuln/search/results?form_type=. [Accessed: 28-Feb-2023].

[13] BSI, "Lagebericht IT-Sicherheit 2022," BSI, 2022. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lag [Accessed: 28-Feb-2023].

[14] L. Erlenhov, F. Gomes de Oliveira Neto, and P. Leitner, "Dependency management bots in open-source systems—prevalence and adoption," PeerJ Computer Science, vol. 8, p. e849, 2022.

[15] L. Erlenhov, "Mapping a Landscape of Developer Assisting Software Bots," Chalmers Tekniska Hogskola (Sweden), 2021.

[16] R. He et al., "Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot," arXiv preprint arXiv:2206.07230, 2022.

[17] M. Wessel et al., "The power of bots: Characterizing and understanding bots in oss projects," Proceedings of the ACM on Human-Computer Interaction, vol. 2, no. CSCW, pp. 1–19, 2018.

[18] security.snyk.io. (accessed on March 22, 2023).

[19] Sonatype, Inc., "State of the software supply chain," 2021. [Online]. Available: https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021. [Accessed: 22-Mar-2023]

[20] T. Winters, T. Manshreck, and H. Wright, Software Engineering at Google: Lessons Learned from Programming over Time. O'Reilly Media, 2020.

[21] R. G. Kula et al., "Do developers update their library dependencies? - an empirical study on the impact of security advisories on library migration," Empir. Softw. Eng., vol. 23, no. 1, pp. 384–417, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5.

[22] E. Derr et al., "Keep me updated: An empirical study of third-party library updatability on android," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, 2017, pp. 2187–2200. [Online]. Available: https://doi.org/10.1145/3133956.3134059.

[23] Dependabot feedback. [Online]. Available: https://github.com/dependabot/dependabot-core/issues/4146, Jan. 2022. [Accessed: 09-April-2023]

[24] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 84–94. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115621.

[25] B. Rombaut, J. Zaytsev, P. Lebleu, and C. Combemale, "There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 1, pp. 1-40, 2023.

[26] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "A model to understand the building and running inter-dependencies of software," in 14th Working Conference on Reverse Engineering (WCRE 2007), Oct. 2007, pp. 140-149

[27] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in Proceedings of the IEEE/ACM International Conference

on Automated Software Engineering, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 309-312. doi: 10.1145/1858996.1859058

[28] GitHub, "Dependabot updates hit GA in GHES," GitHub Blog, 9-Jun-2022. [Online]. Available: https://github.blog/2022-06-09-dependabot-updates-hit-ga-in-ghes/. [Accessed: 22-Mar-2023]

[29] Snyk, "Snyk Community," [Online]. Available: https://snyk.io/community/. [Accessed: 22-Mar-2023]

[30] GitHub, "renovatebot/renovate," [Online]. Available: https://github.com/renovatebot/renovate/discussions. [Accessed: 22-Mar-2023]

[31] Dependabot, "dependabot-core/issues," GitHub. [Online]. Available: https://github.com/dependabot/dependabot-core/issues. [Accessed: 22-Mar-2023]

[32] SNYK. (n.d.). Snyk, "Snyk pricing plans," [Online]. Available: https://snyk.io/de/plans/. [Accessed: 22-Mar-2023]

[33] Snyk, "Continuous Security with Snyk," [Online]. Available: https://security.snyk.io. [Accessed: 22-Mar-2023]

[34] Snyk Ltd., "Snyk Code | GitHub Integration," [Online]. Available: https://snyk.io/code-checker/. [Accessed: 22-Mar-2023]

[35] JFrog, "Pricing," [Online]. Available: https://jfrog.com/pricing/. [Accessed: 22-Mar-2023]

[36] "GitHub Action - Submodule Updates," GitHub Marketplace. [Online]. Available: https://github.com/marketplace/actions/github-action-submodule-updates. [Accessed:22-April-2023].

[37] GitHub Marketplace, "Checkout Submodules," GitHub, 2021. [Online]. Available: https://github.com/marketplace/actions/checkout-submodules. [Accessed: 10-Jul-2023].

[38] G. Saake and K.-U. Sattler, "Algorithmen und Datenstrukturen: Eine Einführung mit Java", 5th ed., dpunkt. verlag, 2014.

[39] A. Himme, "Gütekriterien der Messung: Reliabilität, Validität und Generalisierbarkeit," Methodik der empirischen Forschung, pp. 485-500, 2009.

[40] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms," 3rd ed., ch. 22.3, p. 603, 2009.

[41] Boehm, B. (1999). Managing software productivity and reuse. Computer, 32(9), 111-113.

[42] M. Griss, T. Biggerstaff, S. Henry, I. Jacobson, D. Lea, and W. Tracz, "Systematic software reuse (panel session) objects and frameworks are not enough," ACM SIGPLAN Notices, vol. 30, no. 10, pp. 281-282, 1995.

[43] M.T. Baldassarre, A. Bianch, D. Caivano and G. Vissaggio, "An industrial case study on reuse oriented development," in Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 283-292.

[44] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," Empir. Software Eng., vol. 12, pp. 471-516, 2007.

[45] A. Tomer, L. Goldin, T. Kuflik, E. Kimchi, and S. R. Schach, "Evaluating software reuse alternatives: a model and its application to an industrial case study," IEEE

Transactions on Software Engineering, vol. 30, no. 2, pp. 86-98, Feb. 2004.

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Introduction to Algorithms,” 3rd ed., ch. 22.3, pp. 613-617, 2009.

[47] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Introduction to Algorithms," 3rd ed. Cambridge, MA, USA: MIT Press, 2009, pp. 541-548.

[48] P. Bourhis, N. Bruno, W. Martens, and T. Tan, "JSON: data model, query languages and schema specification," in Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems, 2017, pp. 123-135.

[49] Lois-Guillaume Morand, "Github Action - A practical guide," Packt Publishing, Birmingham, UK, 29-07-2022, p. 12.

[50] MARTIN, R. C. "Design principles and design patterns." Object Mentor, vol. 1, no. 34, 2000, pp. 597.

[51] Maven. (n.d.). Introduction to the Dependency Mechanism. Retrieved April 12, 2023, from https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

[52] Shahin, M., Babar, M. A., Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access, 5, 3909-3943. doi: 10.1109/ACCESS.2017.2684545

[53] "About npm," [Online]. Available: https://docs.npmjs.com/about-npm. [Accessed: Apr. 22, 2023].

[54] Balliauw, M., Decoster, X., Handley, J., Nguyen, L. (2014). Pro NuGet (2nd ed.). Apress.

[55] Ludewig, J. and Lichter, H., "Software Engineering: Grundlagen, Menschen, Prozesse, Techniken," dpunkt.verlag, 3rd ed., 2013.

[56] Continuous Integration vs. Continuous Delivery vs. Continuous Deployment. (n.d.). Atlassian. Retrieved April 22, 2023, from https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[57] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM Journal on Computing, vol. 1, no. 2, pp. 146-160, 1972.

[58] E. Nuutila and E. Soisalon-Soininen, "On finding the strongly connected components in a directed graph," Information Processing Letters, vol. 49, no. 1, pp. 9-14, 1994.

[59] J. Zhou and M. Müller, "Depth-first discovery algorithm for incremental topological sorting of directed acyclic graphs," Information Processing Letters, vol. 88, no. 4, pp. 195-200, 2003.

[60] J. Goll and J. Goll, "Systementwurf bei aspektorientierter Programmierung. Methoden und Architekturen der Softwaretechnik," in Proceedings of the 10th Workshop Software-Reengineering (WSR), Bad Honnef, Germany, 2011, pp. 885-900.

List of Figures

2.1 Bhs corporate statistics 3

3.1 7-point Likert scale 8

4.1 Development of versions and variants 11
4.2 agile V-Modell of 'DigitalEngineering' department 12
4.3 model for reuse by Tomer et al. (2004) 13
4.4 Documentation to inventory in the old mechanism 14
4.5 abstraction of old mechanism 15

6.1 General JSON structure 24
6.2 Cycle inside graph 28

7.1 Sequence diagram of the central method 31
7.2 Integration submodule 32
7.3 general json structure 35
7.4 General dependency type class 36
7.5 Example graph 37
7.6 Cycle detected and resolved 40

8.1 Testing pyramid 44

A.1 Complete workflow 59

List of Tables

4.1 Cost comparison for different savings 15

6.1 Complexity of Graph perations ([38]) 26

7.1 Graph example for Maven citation 38
7.2 Before reverseDependencyMap() 39
7.3 After reverseDependencyMap() 39
7.4 Modification with chain reaction 40
7.5 Modification with chain reaction 42

A.1 List of survey questions and their corresponding text. 55

Appendix A

Rohdaten

A.1 Statistical methods

Definition A.1.1 (Shapiro-Wilk-Test) *To check whether certain data is normally distributed, the Shapiro-Wilk test was performed. The test checks whether a sample is drawn from a normal distribution.*

Definition A.1.2 (Pearson-Test) *The Pearson Chi-Square test tests whether there is a relationship between two categorical variables. The observed frequencies are compared with theoretically expected frequencies. Afterwards, the strength and direction of the relationship are determined.*

Definition A.1.3 (Spearman-Rho) *The Spearman rank correlation analysis calculates the linear relationship between two at least ordinal-scaled variables. Since the relationship between two variables is always being investigated, it is referred to as a "bivariate relationship".*

Definition A.1.4 (Phi-Koeffizient) *The Phi coefficient (also called Phi correlation coefficient) is a measure of the strength of the relationship between two dichotomous variables. The Phi coefficient has a range of values from -1 to 1, where a value of 0 means that there is no association between the variables, a value of 1 means a perfect positive association, and a value of -1 means a perfect negative association.*

Definition A.1.5 (Cramer V) *Cramer's V is an extension of the Phi coefficient for tables with more than two categories per variable. It measures the strength of the relationship between two variables in a cross table with more than two categories per variable. The value of Cramer's V ranges from 0 to 1, where a value of 0 means that there is no association and a value of 1 means a perfect association.*

A.2 Statistical evaluation of the questionnaire

Coding of the questions

The questions are coded according to this scheme:

Code	Question
F10	What overall 'Issue Types' do you use? (e.g. feature/bug/documentation/refactoring)
F11	What is your typical pull/push cycle from your local repository to GitHub?
F12	How often does your Code get merged to the MAIN branch (on average)?
F13	When connecting libraries (or similar external code fragments) to your program what type of version control are you most likely to use?
F15	How much do you agree with the following statement?
F15_3	I create simple but meaningful comments
F15_13	Most of my Classes have only one responsibility

Table A.1: List of survey questions and their corresponding text.

A.2.1 Question (F10) Push-Pull-cycles

For the distribution with 12 participants and categories 1-8, the calculation of the Shapiro-Wilk test (p = .013 < .05) showed that there is no normal distribution. Due to the non-normal distribution and the low number of observations in some categories, statistical tests are not informative. However, there appears to be a clear preference for a "daily multiple push-pull cycle" among the participants.

Question (F11) Merge-To-Main-cycles

The calculation of the Shapiro-Wilk test (p = .013 < .05) indicated that the distribution is not normal. As the distribution is not normally distributed and the number of observations in some categories is very small, statistical tests are not meaningful. The comments indicate that the number of merge operations to the main codebase varies depending on the situation, but more than half of the participants perform at least one merge from other branches to the main codebase once a week. This suggests that other branches are also used and the main codebase is not the sole working branch
Spearman-Rho Korrelation (F11 und F12) -> Abbildung X zeigt, dass die Korrelation zwischen F11 und F12 bei rs = .643 liegt. Der p-Wert beträgt .284. Somit ist die Korrelation statistisch nicht signifikant (p < .05).

F10 and F13

As can be seen in Figure X, both Cramer's V (.676) and Phi (.955) are significant (both p < .001). Since the values are above .50, a very strong association is assumed.

F15

For F15, participants were asked about their personal assessment of their implementation of object-oriented principles. The questions could be answered on a scale from "not at all" to "Agree completely" (1-7). There are six questions of interest in this regard.

A Pearson correlation table was created. Only F15_9 and F15_10 are not significantly correlated (N = 14, p = .054 > .05). All other combinations show a significant strong association. Due to non-responses to questions, missing answers were supplemented with the respective average of the question to maintain a consistent number of votes.

A.3 Example output of Write2Inventory

```
{
  "repository": {
    "orgName": "DigitalEngineering",
    "repoName": "DigitalEngineering/DE_ID_AC_Dependencies2Inventory",
    "currentReleaseTag": "v0.0.16",
    "license": "MIT License",
    "sha": "43b9c2b7cbe36d97b6e61a6361f39df0e2f8dd8b"
  },
  "InternNpmPackages": [
    {
      "project": "@digitalengineering/de_id_ac_dependencies2inventory/Testanwendung",
      "source": "https://npm.github.bhs-world.com/download/@digitalengineering/...",
      "name": "@digitalengineering/de_id_np_installdependencies",
      "currentVersion": "0.6.2",
    }
  ],
  "ExternNpmPackages": [
    {
      "project": "@digitalengineering/de_id_ac_dependencies2inventory",
      "source": "https://registry.npmjs.org/@actions/core/-/core-1.10.0.tgz"
      "name": "@actions/core",
      "currentVersion": "1.10.0",
    },
    ...
  ],
  "InternNugetPackages": [
    {
      "project": "@digitalengineering/de_id_ac_dependencies2inventory/Blazor/...",
      "source": "https://nuget.github.bhs-world.com/digitalengineering/...",
      "name": "Bhs.Design",
      "currentVersion": "2.5.3",
    }
  ],
  "ExternNugetPackages": [
    {
      "project": "@digitalengineering/de_id_ac_dependencies2inventory/Blazor/...",
      "source": "https://api.nuget.org/v3/index.json",
      "name": "Microsoft.EntityFrameworkCore.SqlServer",
      "currentVersion": "1.10.0",
    }
  ],
  "InternSubmodules": [
    {
      "name": "DE_TP_Documentation",
      "url": "docs/DE_TP_Documentation"
      "sha": "bbe00b931dfe89878a2175f8f3737e60888bf1aa",
    }
  ],
  "ExternSubmodules": [],
  "updateStrategy": "MINOR"
}
```

A.4 Whole workflow

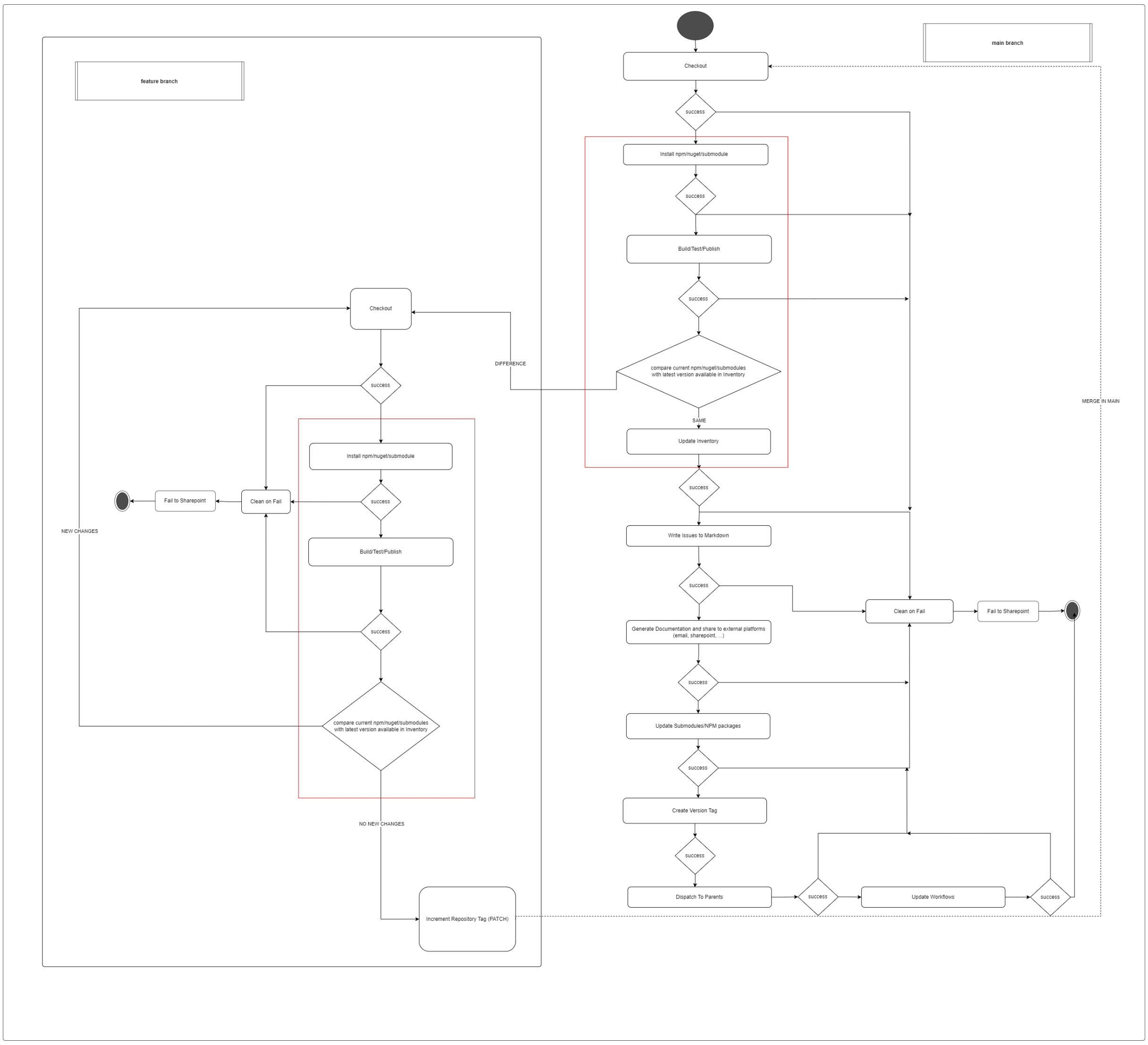


Figure A.1: Complete workflow