D2.3.1 Documented data analysis examples

Contents

1. Executive summary 3
2. Data fetching and modelling 4
   a. Technical Debt Data Fetching and Data Model 4
      i. Collecting data from Issue Trackers 4
      ii. Filtering issues 5
      iii. Linking issues with commits 5
      iv. Data model for Self-Admitted Technical Debt in issues 5
   b. Product Quality Data Fetching and Data Model 6
      i. Collecting data from Static Analysis Tool 6
      ii. Data model for Product Quality 7
3. Data analysis examples per use case 9
   a. Quality Use Case: Technical Debt 9
   b. Quality Use Case: Product Quality 15
   c. Quality Use Case: Runtime Performance 18
   d. Software as a Service (SaaS) Use Case 19
      i. Background 19
      ii. Goal 20
      iii. Data model 20
      iv. Integrations 22
      v. Visualizations 22
   e. Teaching Use Case 23
      i. Background 23
      ii. User study – needs for analysis 24
      iii. Requirements for analysis 24
         1) Overview and background 24
         2) Stakeholder interests 26
         3) Scenarios 26
      iv. Analyses provided by visualization demo 27
4. Conclusions 35
5. References 36

REVISION HISTORY			
Version	**Date**	**Author/ Organization**	**Modifications**
0.1	01/12/2019	Paris Avgeriou (RUG)	Drafted outline and executive summary
0.2	01/05/2020	Paris Avgeriou and Yikun Li (RUG)	Data analysis examples of Technical Debt use case
0.3	04/06/2020	Lidia López (UPC)	Data analysis example for Product quality use case
0.4	15/06/2020	Paris Avgeriou (RUG) Lou Somers (CPP)	CPP analysis example, re-structuring document and resolving comments
0.5	02/09/2020	Lidia López (UPC)	Product Quality case reviewed
0.6	10/09/2020	Paris Avgeriou (RUG)	Consolidation and refinement of text
0.7	10/09/2020	Outi Sievi-Korte and Vivian Lunnikivi (TAU)	Added Teaching use case
0.8	14/09/2020	Paris Avgeriou (RUG) Lidia López (UPC)	Conclusions added and references reviewed
0.9	17/09/2020	Veli-Pekka Eloranta (Vincit)	Added SaaS use case
1.0	29/9/2020	Paris Avgeriou and Yikun Li (RUG)	Revision based on comments from reviewers

1. Executive summary

The data that are illustrated in VISDOM visualizations originate from a number of heterogeneous sources, across the DevOps tool chain. In addition, a significant amount of processing is required before it can be used in the visualizations implemented by VISDOM. The goal of WP2 is to deliver the data to the visualization techniques and tools (WP3), based on the requirements and use cases that have been defined in WP1. Task 2.3 focuses on data analysis and metrics and complements the other tasks in WP2 that focus on techniques and methods to acquire (Task 2.1), and model (Task 2.2) data from software project repositories and product artifacts. Task 2.3 results in two deliverables: D2.3.1 (the current deliverable) which focuses on examples of data analysis, and D2.3.2 which focuses on data metrics.

One of the core ideas behind VISDOM is to analyse the data from a number of heterogeneous sources in order for them to be effectively visualised in dashboards. Furthermore, in WP1 we have defined three different use cases, where each one focuses on a different aspect or application domain (software quality, teaching and Software as a Service - SaaS), and hence tools and corresponding data. Therefore, in this deliverable, we will describe for each use case, examples of customized analysis approaches for its respective data. Accordingly, this deliverable is organized according to the three use cases. The first use case that focuses on Software Quality is more comprehensive than the other two as it comprises a number of different aspects: technical debt, runtime performance and product quality. Each one of these three aspects is elaborated within a different subsection.

The five analysis examples are developed and presented independently in this deliverable. However, there is a number of common aspects across the use cases; for example technical debt is complementary to product quality, while they both can be used in the SaaS use case and the Teaching use case. Such synergy and the corresponding integration between the analysis tools are planned as future work within WP3 (see Section 4).

The rest of this deliverable is structured as follows. Section 2 presents examples of data fetching and modelling. This has two purposes: first, it links this deliverable to Tasks 2.1 and 2.2 that concern data fetching and modelling, as analysis is *based on* fetching and modelling; second it acts as background knowledge to facilitate understanding of the data analysis models presented in this deliverable. Section 3 is the core of this deliverable and elaborates on the data analysis examples. As aforementioned, the examples are categorized according to the three use cases of the VISDOM project. Five examples are discussed in detail: three for the quality use case and two for the teaching and SaaS use cases. Finally, Section 4 concludes this deliverable and briefly outlines current and future work.

2. Data fetching and modelling

Data analysis can proceed as soon as data are fetched (Task 2.1) and properly modelled (Task 2.2). Deliverable D2.1.1 has elaborated on the state of the art on data fetchers (requirements and architecture) for different DevOps tools and tool-chains. We have implemented a number of data fetchers for each of the use cases investigated in this project. Deliverable D2.2.1 is currently work in progress and contains documentations of the data models used in the respective use cases. There are currently a number of data models that focus on different aspects of the DevOps tool chain, such as models on technical debt or runtime qualities.

In the following sub-sections we provide two examples on both the data fetching and modelling for the Quality Use Case. These act as a precursor of the deliverable D2.2.1 and are meant to facilitate the comprehension of the data analysis examples presented in the current deliverable. Particularly, we show an example for Self-Admitted Technical Debt in Issue Trackers and a second example for the Product Quality; in these two examples, data are fetched from Issue Trackers, Static Code Analysis tools and Continuous Integration tools.

a. Technical Debt Data Fetching and Data Model

Technical debt (TD) refers to taking shortcuts, either deliberately or inadvertently, to achieve short-term goals, which might negatively influence the maintenance and evolution of software in the long term [1]. A part of technical debt is declared as such by the developers themselves; for example when developers state in source code comments, that something is not right and should be fixed. This has been termed “Self-Admitted Technical Debt” (SATD) [2]. In this example we work on analysing the types of SATD in issue tracking systems. The next sub-sections elaborate on how the relevant data are fetched and modelled while Section 3.a will elaborate on the actual analysis.

i. Collecting data from Issue Trackers

To collect issue data, we looked into Apache Java projects since they are of high quality and supported by mature communities. To select Apache projects pertinent to our study goal, we set the following criteria:

- Both the issue tracking project and the source code repository are publicly available and well-maintained.
- They have at least 1,000,000 source lines of code (SLOC) and 10,000 issues in the issue tracker. This is to ensure sufficient complexity.
- Source code commits involve their associated issue keys within their comments. This is important to support linking commits (in the source code repository) with issues (in the issue tracker).
- They are commonly used in other self-admitted technical debt studies [3]. This allows us to compare the results between our study and other self-admitted technical debt studies.

Based on these criteria, we selected Hadoop[1] and Camel[2]. Both projects were studied for self-admitted technical debt [3], were developed in Java, used Git as a source code repository and JIRA[3] as an issue tracker. We analysed the latest released versions on Jan 16, 2020. Table 2.a.I. shows some details for the two projects. The number of Java files and SLOC are calculated using

[1] https://hadoop.apache.org

[2] https://camel.apache.org

[3] https://jira.apache.org

the LOC tool[4]. The number of contributors is obtained from GitHub. We used the JIRA Python package to extract all Hadoop and Camel issues from the online server and stored them in a local database; then we counted the number of issues.

TABLE 2.a.I.

DETAILS OF CHOSEN PROJECTS.

Project	# Java files	SLOC	# Contributors	# Issues	# Filtered issues
Hadoop	10,918	1,700,501	259	16,808	6,685
Camel	17,585	1,196,790	583	14,411	12,259

ii. Filtering issues

To ensure that we study issues with a complete life cycle, we applied two filtering criteria:

1. Issue status: Since we are aiming at studying technical debt items that were resolved, we focus on issues that are done or closed. Thus, we removed all issues with status Open or Pending Closed.
2. Availability of issue key in commits: Although some issues have their status set to Resolved and developers commented that the patches are successfully committed to the repositories, we cannot find the related commits in Git. This is mostly because developers did not include the issue key in the corresponding commit messages. We also exclude these issues, since we need the commit information to be able to answer the research question on debt repayment.

The final number of issues after filtering is listed in the rightmost column of Table 2.a.I.

iii. Linking issues with commits

In order to determine how software engineers actually resolve technical debt, we have to capture the code commits associated with an issue. This information is needed to determine the software developers responsible for repaying technical debt and the time for this repayment.

Since in the previous step, we ensured that the commit messages contain the related issue keys, we use those keys to link issues with commits. In practice, we first output the Git commit log, and match the issue key by applying a regular expression to the commit log. Then all matched commits (including commit date, commit message, and commit author) are inserted into the issue holding the issue key ordered by time, and then the issue with commit information is stored in a local database.

iv. Data model for Self-Admitted Technical Debt in issues

We have used an existing framework from Alves et al. [4] to model the Technical Debt types. This framework provides basic types of technical debt, with high-level definitions and a list of indicators per type. Table 2.a.II presents the indicators for each Technical Debt type, as well as their definitions.

TABLE 2.a.II.

DATA MODEL OF TECHNICAL DEBT TYPES AND CORRESPONDING INDICATORS (ADAPTED FROM [4]).

[4] https://github.com/cgag/loc

Type	Indicator	Definition
Architecture Debt	Violation of modularity	Because shortcuts were taken, multiple modules became inter-dependent, while they should be independent.
	Using obsolete technology	Architecturally-significant technology has become obsolete.
Build Debt	Under- or over-declared dependencies	Under-declared dependencies: dependencies in upstream libraries are not declared and rely on dependencies in lower level libraries. Over-declared dependencies: unneeded dependencies are declared.
	Poor Deployment Practice	The quality of deployment is low that compile flags or build targets are not well organised.
Code Debt	Complex code	Code has accidental complexity and requires extra refactoring action to reduce this complexity.
	Dead code	Code is no longer used and needs to be removed.
	Duplicated code	Code that occurs more than once instead of as a single reusable function.
	Low-quality code	Code quality is low, for example because it is unreadable, inconsistent, or violating coding conventions.
	Multi-thread correctness	Thread-safe code is not correct and may potentially result in synchronisation problems or efficiency problems.
	Slow algorithm	A non-optimal algorithm is utilised that runs slowly.
Defect Debt	Uncorrected known defects	Defects are found by developers but ignored or deferred to be fixed.
Design Debt	Non-optimal decisions	Non-optimal design decisions are adopted.
Documentation Debt	Outdated documentation	A function or class is added, removed, or modified in the system, but the documentation has not been updated to reflect the change.
	Low-quality documentation	The documentation has been updated reflecting the changes in the system, but quality of updated documentation is low.
Requirements Debt	Requirement partially implemented	Requirements are implemented, but some are not fully implemented.
	Non-functional requirements not fully satisfied	Non-functional requirements (e.g. availability, capacity, concurrency, extensibility), as described by scenarios, are not fully satisfied.
Test Debt	Expensive tests	Tests are expensive, resulting in slowing down testing activities. Extra refactoring actions are needed to simplify tests.
	Lack of tests	A function is added, but no tests are added to cover the new function.
	Low coverage	Only part of the source code is executed during testing.

b. Product Quality Data Fetching and Data Model

In order to analyse the product quality of a software system, the data is fetched from the following data sources: GitLab[5] (issues), SonarQube[6] (code quality), and Jenkins[7] (test execution results).

i. Collecting data from Static Analysis Tool

To collect software static analysis, we used a concrete project from Experis. Concretely, we collect data from the SonarQube tool, which provides continuous inspection of the source code quality and security, providing a set of metrics that can be used as part of the product quality assessment. Table 2.b.I includes the metrics fetched from SonarQube tool.

TABLE 2.b.I.

SONARQUBE FETCHED DATA FOR EXPERIS USE CASE

Metric ID	Metric Name	Type	Description	SonarQube Domain
classes	Classes	INT	Total number of classes	Size
comment_lines	Comment Lines	INT	Number of comment lines	Size

[5] https://about.gitlab.com/

[6] https://www.sonarqube.org/

[7] https://www.jenkins.io/

comment_lines_density	Comments (%)	PERCENT	Comments balanced by ncloc + comment lines	Size
directories	Directories	INT	Number of directories	Size
lines	Lines	INT	Number of lines	Size
files	Files	INT	Number of files	Size
functions	functions	INT	Number of functions	Size
ncloc	Lines of Code	INT	Non commenting lines of code	Size
complexity	Cyclomatic Complexity	INT	Cyclomatic complexity	Complexity
duplicated_blocks	Duplicated Blocks	INT	Number of duplicated blocks	Duplications
duplicated_files	Duplicated Files	INT	Number of duplicate files	Duplications
duplicated_lines_density	Duplicated Lines (%)	PERCENT	Duplicated lines balanced by statements	Duplications
function_complexity	Complexity / Function	FLOAT	Complexity average by function	Complexity
open_issues	Open Issues	INT	Number of open issues	Issues

ii. Data model for Product Quality

For the assessment of product quality, we defined a set of indicators aggregated in a 3-level quality model [5] (see Fig. 2.b.I). Each quality model level is providing quality assessment at different levels:

- Quality metrics: a metric defines how the quality of a specific attribute of an entity is measured, it provides a means to quantify factors that characterize this entity (or a related one).
- Quality factors: a quality factor constitutes a property of the software product. These quality factors can be assessing a quality related to the software product or software development process. They are computed as an aggregation of quality metrics using a weighted average. For example, “Testing Status” or “Code Quality” as product quality factors, and “Productivity” or “Performance” as process quality factors.
- Strategic Indicators: a strategic indicator is an aspect that a company considers relevant for the decision-making process during the software process development. They are computed as an aggregation of quality factors using an average or a weighted average. For example, time-to-market, maintenance cost, customer satisfaction.

Fig. 2.b.I shows the proof-of-concept data model defined for the Experis case. This initial quality model combines information from all the sources (GitLab, SonarQube, and Jenkins) to define the strategic indicator “*Product Quality*”. All the indicators are computed using an average of the lower level of the model. This strategic indicator includes *product* quality factors covering the DevOps life cycle:

- *Issues quality* in terms of being well-defined in the backlog (plan)
- *Code quality* in terms of source code quality (code)
- *Testing status* in terms of performance and test success (test & build)
- *Software stability* in terms of number of open bugs (monitor)

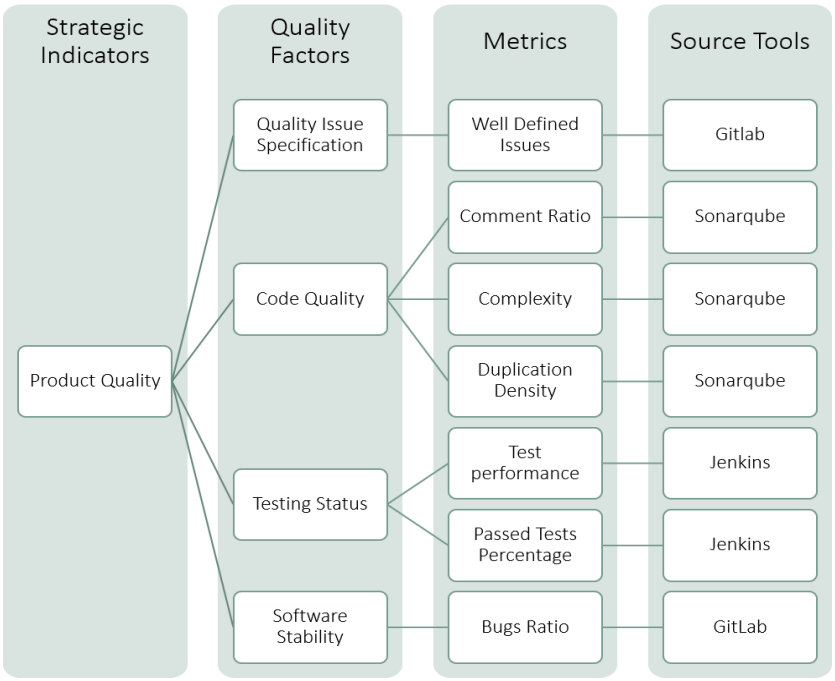


Fig. 2.b.I. Product Quality Data Model

3. Data analysis examples per use case

The next sub-sections elaborate on the data analysis examples for each use case (quality, teaching, Software as a Service - SaaS). As aforementioned, the quality use case is further decomposed into three aspects: Technical Debt, Product Quality, and Runtime Performance (respectively subsections 3.a., 3.b and 3.c).

a. Quality Use Case: Technical Debt

In this example we analyse the types of self-admitted technical debt in issue tracking systems, and determine how software engineers identify and resolve them.

1) What types of technical debt are reported in issue trackers?

We found eight types of technical debt in issue trackers: architecture, build, code, defect, design, documentation, requirement, and test debt. For each type we found one or more indicators (see data model in Section 2.a.iv). In the following paragraphs, we report on the associated indicators for each type, also providing a quote from actual issues to exemplify each indicator.

- **Architecture debt**: problems that are architecturally significant, i.e. they are hard to change. Most of the debt in this type relates to the indicator Violation of Modularity.

  "It would be good if these were moved into their own module...." - [Camel-4543][8]

  Some architecture debt is caused by *Using Obsolete Technology*.

  "The camel-atom component is using an ancient incubator version of abdera which will make it hard to work with camel-cxf." - [Camel-4132]
- **Build debt**: issues that make building (i.e. source code compilation to artifacts) harder or more time-consuming. Most of the identified build debt is caused by Over- or Under-Declared Dependencies.

  "Avoid the redundant direct dependency on log4j by the components." - [Camel-4331]

  The rest of build debt is caused by *Poor Deployment Practice*.

  "Rationalize the way architecture-specific sub-components are built with ant in branch-1. This is a matter of maintainability and understandability, and therefore robustness under future changes in build.xml." - [Hadoop-8364]
- **Code debt**: issues in source code, which negatively influence the maintenance of software. Most of the code debt is caused by *Low-Quality Code*.

  "This will lead to very unmaintainable code. We absolutely do not want to have nested retries for different contexts." - [Hadoop-3198]

  A few code debt items result from *Slow Algorithm*.

  "#query() does O(N) calls LinkedList#get() in a loop, rather than using an iterator. This makes query O(N^2), rather than O(N)." - [Hadoop- 8866]

  Multi-Thread Correctness is another factor causing code debt.

[8] https://jira.apache.org/jira/browse/CAMEL-4543

"EnsureInitialized() forced many frequently called methods to unconditionally acquire the class lock." - [Hadoop- 9748]

The rest of the code debt is caused by Dead Code, Duplicated Code, and Complex Code.

"As we don't use the CxfSoap component any more, it's time to clean it up." - [Camel-2523]

"I am concerned about the code duplication this brings." - [Hadoop- 6381]

"...can be simplified to the following so there aren't so many return statements to track." - [Hadoop-10169]

- **Defect debt**: known defects that are deferred to be fixed. *All defect debt items are caused by Uncorrected Known Defects*.

  "This works in 2.12.x onwards. Hunting this down on 2.11.x is low priority. End users is encourage to upgrade if they really need this." - [Camel-6735]
- **Design debt**: shortcuts or non-optimal decisions taken in detailed design. All design debt results from *Non-Optimal Decisions*.

  "Instead of passing a long[] you should pass a struct that implements Writable." - [Hadoop-481]
- **Documentation debt**: when the software is modified, the documentation is not updated to reflect the changes or the quality of updated documentation is low. Most of this type of debt is caused by *Outdated Documentation*.

  "The maven reports is just getting to old and intermixed with 1.x and trunk releases." - [Camel-1846]

  The second indicator is *Low-Quality Documentation*.

  "I agree to improve documentation to make it clear that..." - [Hadoop-12672]
- **Requirement debt**: when the requirements specification is not in line with the actual implementation. Some requirement debt is caused by *Requirements Partially Implemented*.

  "The only feature which we don't support is correlated message groups. That requires a bit more work and also may complicated..." - [Camel-1669]

  Another common cause concerns *Non-Functional Requirements Not Being Fully Satisfied*. In the example below, concurrency is not fully satisfied.

  "Definition requires the implementations for its interfaces should be thread-safe. HarFsInputStream doesn't implement these interfaces with tread-safe, this JIRA is to fix this." - [Camel-5587]
- **Test debt**: shortcuts or non-optimal decisions taken in testing that negatively affect maintainability. Most test debt is caused by *Lack of Tests*.

  "There are no XQuery specific tests." - [Camel-201]

  The other major cause of test debt is *Low Coverage*.

"Some of the test code doesn't check for correct error codes to correspond with the wrapped exception type." - [Hadoop-11103]

Finally, some test debt results from *Expensive Tests*.

"I see recent hadoop-hdfs test runs have been taking 2.5 hours. This one (new patch) was 45 minutes." - [Hadoop-11670]

Table 3.a.I. presents an overview of technical debt types and indicators in the examined issues. We observe that code, documentation, and test debt are the three most common types (with 38.8%, 21.7%, and 18.4% respectively). Furthermore, the three most common indicators are *Low-quality Code*, *Lack of Tests*, and *Outdated Documentation*.

TABLE 3.a.I.

TYPES AND INDICATORS OF TECHNICAL DEBT.

Type	Indicator	#	#	%
Architecture debt	Violation of modularity	8	10	6.6
	Using obsolete technology	2		
Build debt	Over- or under-declared dependencies	5	6	3.9
	Poor deployment practice	1		
Code debt	Complex code	2	59	38.8
	Dead code	12		
	Duplicated code	6		
	Low-quality code	36		
	Multi-thread correctness	1		
	Slow algorithm	2		
Defect debt	Uncorrected known defects	4	4	2.6
Design debt	Non-optimal decisions	8	8	5.3
Documentation debt	Low-quality documentation	16	33	21.7
	Outdated documentation	17		
Requirements debt	Requirement partially implemented	3	4	2.6
	Non-functional requirements not being fully satisfied	1		
Test debt	Expensive tests	1	28	18.4
	Lack of tests	20		
	Low coverage	7		

Finally, since we annotated technical debt on the sentence level (instead of the issue level), an issue may contain more than one type of technical debt. Table 3.a.II. presents how many issues contain zero, one or more types of technical debt in issues. As we can see, 24 out of 117 issues (20%) that contain technical debt, contain more than one type. This validates our choice to analyse issues at the level of sentences; if we had performed the analysis at the level of issues, we would have missed the additional technical debt types per issue.

TABLE 3.a.II.

NUMBERS OF TYPES OF TECHNICAL DEBT IN ISSUES.

Issue description	# Issues	% Issues
Does not contain technical debt	383	76.6
Contains one type of technical debt	93	18.6
Contains two types of technical debt	21	4.2
Contains three types of technical debt	2	0.4
Contains four types of technical debt	1	0.2

2) When do software engineers identify technical debt?

We observed three distinct cases of technical debt being identified in issue trackers:

1. *Identifying technical debt before creating an issue (i.e. debt is the reason for creating the issue):* When developers spot an existing technical debt item in the system, they report it in an issue tracker to be resolved. For instance, a developer found low-quality code, which complicates debugging; thus, he/she created a new issue:

   "If the user doesn't setup the right camel context for the context component. The exception we got is misleading, we need to throw more meaningful exception for it." - [Camel-5714]

2. *Identifying technical debt during code review:* As explained in Section III, software engineers perform code reviews by creating and reviewing code patches in issue trackers. When a code reviewer identifies technical debt items in a code patch, he/she discusses it with other developers to determine, if the identified technical debt should be resolved or committed to the system. For example, during a code review, a developer found that a shortcut was taken. Thus, he/she commented on a patch:

   "The patch looks good to me... It would be better if we can add an upper limit for the size of the GSet." - [Hadoop-9763]

3. *Identifying technical debt after a patch is committed*: Technical debt can exist in a patch but go undetected through the code review; after the patch is committed, a developer may notice the debt in the commit and report it. For instance, after a command patch is committed to the repository, a developer noticed that documentation is not updated accordingly:

   "We need to update the documentation with the new command." - [Camel-8101]

TABLE 3.a.III.

TECHNICAL DEBT IDENTIFICATION CASES.

Project	# Identified	Case 1		Case 2		Case 3	
		#	%	#	%	#	%
Hadoop	101	41	40.6	57	56.4	3	3.0
Camel	51	27	52.9	13	25.5	11	21.6
Total	152	68	44.7	70	46.1	14	9.2

To gain a better understanding of how technical debt is identified, Table 3.a.III. presents the count of technical debt items for the three aforementioned cases. Clearly, the first and second cases represent the majority (44.7% and 46.1% respectively) in these projects. Compared with Camel, there is 30.9% more debt introduced in Hadoop with the second case and 18.6% less debt introduced with the third case. This means that more technical debt is identified during code

reviews (on patches) than after the patch is committed in the system in Hadoop compared with Camel.

Moreover, we also investigate who reported the debt: the developers who created it in the first place or those who discovered it. Since technical debt identified in the first case already exists in the system, information on who created it is not contained in issue trackers; thus, such information is obtained only for technical debt identified in the second and third cases. Table 3.a.IV. presents an overview on who reported the technical debt. We find that on average most of the debt is reported by other developers (i.e. 86.9%), and a small part is self-reported (reported by those that created it). Camel has a higher percentage of self-reported debt than Hadoop, but the vast majority of its debt is still reported by others (i.e. 70.8% versus 29.2%). This may mean that most developers create technical debt unintentionally.

TABLE 3.a.IV.

TECHNICAL DEBT REPORTERS.

Project	Reported by creators		Reported by others	
	#	%	#	%
Hadoop	4	6.7	56	93.3
Camel	7	29.2	17	70.8
Total	11	13.1	73	86.9

3) How do software engineers resolve technical debt within issues?

We first analyse how much technical debt is paid off. Table 3.a.V. presents the amounts and percentages of technical debt items that are identified and resolved. We can see that most of the identified technical debt is actually resolved in both Hadoop and Camel (i.e. 71.3% and 72.5%, respectively). This indicates that, when technical debt is reported in issue trackers, it will likely be resolved. In other words, most software developers are conscious of the importance of paying off technical debt items.

TABLE 3.a.V.

AMOUNT OF TECHNICAL DEBT THAT WAS REPAID.

Project	# Identified	# Repaid	% Repaid	% Remaining
Hadoop	101	72	71.3	28.7
Camel	51	37	72.5	27.5
Total	152	109	71.7	28.3

The next question we answer is who repays technical debt. As shown in Table 3.a.VI., we distinguish between developers who create technical debt, those who identify it and other developers who participate in resolving it. We can see that most of the technical debt is repaid by those who identified it (i.e. 47.7%), and those who created it (i.e. 44.0%); while only 8.3% debt is resolved by other developers. This shows that developers take the responsibility to pay off most of the technical debt they identified or created themselves.

TABLE 3.a.VI.

WHO REPAID TECHNICAL DEBT.

Project	# Repaid	# Repaid by					
		Creators		Identifiers		Others	
		#	%	#	%	#	%
Hadoop	72	36	50.0	33	45.8	3	4.2
Camel	37	12	32.4	19	51.4	6	16.2
Total	109	48	44.0	52	47.7	9	8.3

The final question that we answer is how long it takes to fix technical debt. Fig. 3.a.I shows the mean times, the median times, and the time distributions of technical debt repayment for the two projects. With a visual inspection, we see that the time spent to fix technical debt in Hadoop and Camel varies. We also observe that after the technical debt is reported (point zero in the y axis), most of the fixes happened in a short time compared to the average (67.0% of the debt is repaid in the first 100 hours).

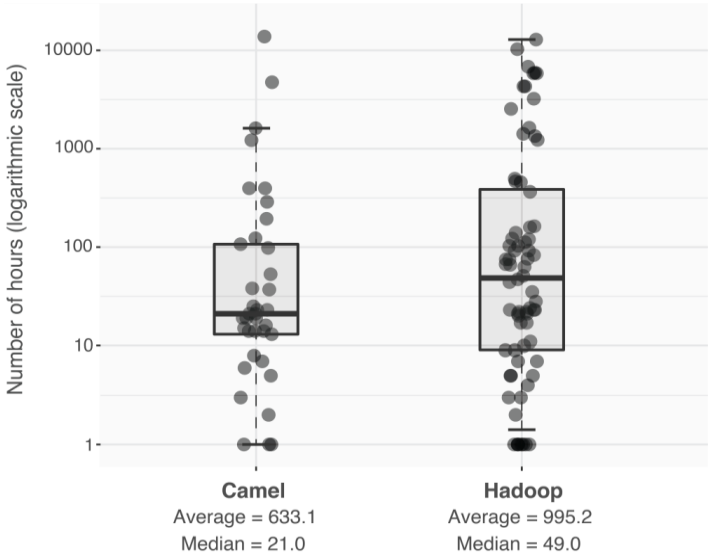


Fig. 3.a.I. The time distribution of technical debt repayment in issue trackers.

Furthermore, we compare the time spent on resolving technical debt by different developers. More specifically, we compare repayment time distributions between pairs of developers (e.g. between creators and identifiers) using the Mann-Whitney test [6] and Cliff's delta [7] to determine the significance level and the effect size of the differences. The result is demonstrated in Table 3.a.VII. There are notable differences between Hadoop and Camel. In Hadoop, the repayment time of identifiers and others is longer than creators with statistical significance (*p*-values are 0.031 and 0.028 respectively). Moreover, the time difference between identifiers and others is at the margin of statistical significance (*p*-value is 0.080). According to the effect size, we observe that the difference between creators and identifiers is small, while the difference between identifiers and others is large. Thus, technical debt in Hadoop is paid back the quickest by creators, followed with a small margin by identifiers, followed with a large margin by others. In Camel, the situation is different as none of the time differences is statistically significant. We only observe that the repayment time of others is much longer (on average) than creators and identifiers.

TABLE 3.a.VII.

REPAYMENT TIME COMPARISON BETWEEN DIFFERENT DEVELOPERS.

Project	Average time spent on debt repayment (h)		*p*-value	Cliff's delta
	Creators	Identifiers		
Hadoop	128.0	1510.8	0.031	-0.303 (small)
Camel	174.5	142.3	0.935	0.021 (negligible)
	Identifiers	**Others**		
Hadoop	128.0	5730.3	0.028	-0.777 (large)
Camel	174.5	3104.3	0.851	-0.069 (negligible)
	Identifiers	**Others**		
Hadoop	1510.8	5730.3	0.080	-0.626 (large)
Camel	142.3	3104.3	0.463	-0.210 (small)

b. Quality Use Case: Product Quality

The product quality data is continuously fetched and the quality model (see Section 2.b.ii) is used to continuously assess product quality in order to visualise quality. The software analytics tool Q-Rapids [8], is used in this section to illustrate the analysis results.

All the quality model indicators are computed to be interpreted as a bad/good quality, so the results of the analysis are values for each one between 0 and 1, with 0 being interpreted as the worst quality values and 1 the best quality. In order to support decision-makers, the quantitative assessment (values from 0 to 1) are classified as categories to provide a qualitative assessment (from bad to good). In the proof of concept, all the indicators (metrics, factors, and strategic indicators) have been classified in three equally distributed categories: Bad, [0, 0.33); Neutral, [0.33, 0.66); Good, [0.66, 1].

As a proof-of-concept, we analysed two pilot projects: PHE and PHE_server, which belong to a single product called PHE. It consists of a mobile application aimed at improving the welfare state in the workplace. PHE contains the development related to the mobile application, while PHE_server contains the database and the intelligence offered by the application, meaning that it is related to the operation phase. The tools used by the development team, which are used to collect the raw data, are: GitLab, SonarQube, and Jenkins. The product quality model described in Section 2.b (Fig. 2.b.I) has been customised for each project to fit with the produced data. Fig. 3.b.I shows the visual representation of the instance of the project quality model applied to project PHE on July 31st, 2020. For the *Product Quality* strategic indicators, we implemented metrics related to *Code Quality* factor (*Comment Ratio* and *Duplication Density*) and some metrics related to the development process (*Software Readiness* strategic indicator). Fig 3.b.II shows the quality visualization for project PHE at the level of strategic indicators. Both indicators quality assessment correspond to the neutral zone.

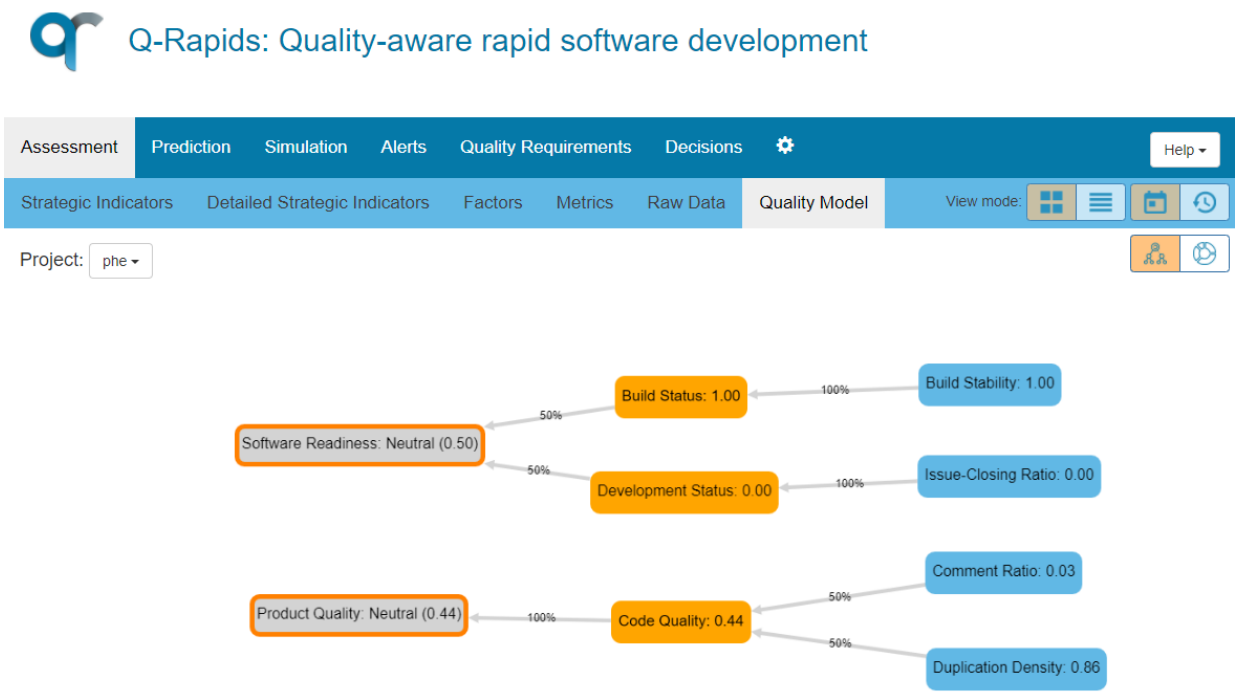
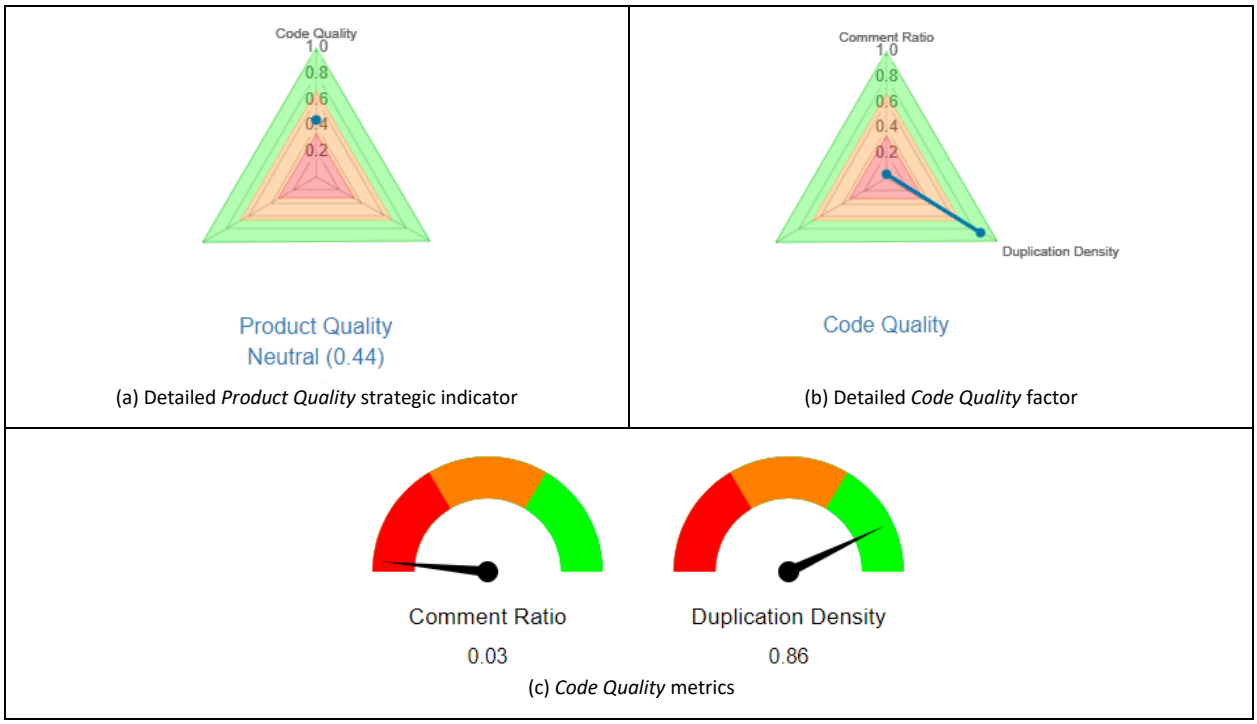


Fig. 3.b.I.PHE project Quality Assessment.

Fig. 3.b.II.PHE Quality Assessment Visualization.

Q-Rapids includes views for the different levels in the quality model to allow decision-makers to understand the assessment. Fig. 3.b.III shows the different views to understand *Product Quality* strategic indicator assessment. Fig 3.b.III (a) shows that there is only one factor impacting Product Quality (*Code Quality*), and its assessment is 0.44 (neutral zone); (b) shows that Code Quality Factor is impacted by two metrics (*Comment Ratio* and *Duplication Density*), this chart allows to see that the problematic metric is related to commented code, duplication is quite good; (c) visualises metrics independently.



(a) Detailed *Product Quality* strategic indicator

(b) Detailed *Code Quality* factor

(c) *Code Quality* metrics

Fig. 3.b.III.PHE Product Quality assessment rationale.

Fig 3.b.IV includes the textual view for metrics, including information about the metric description and the assessment computation.

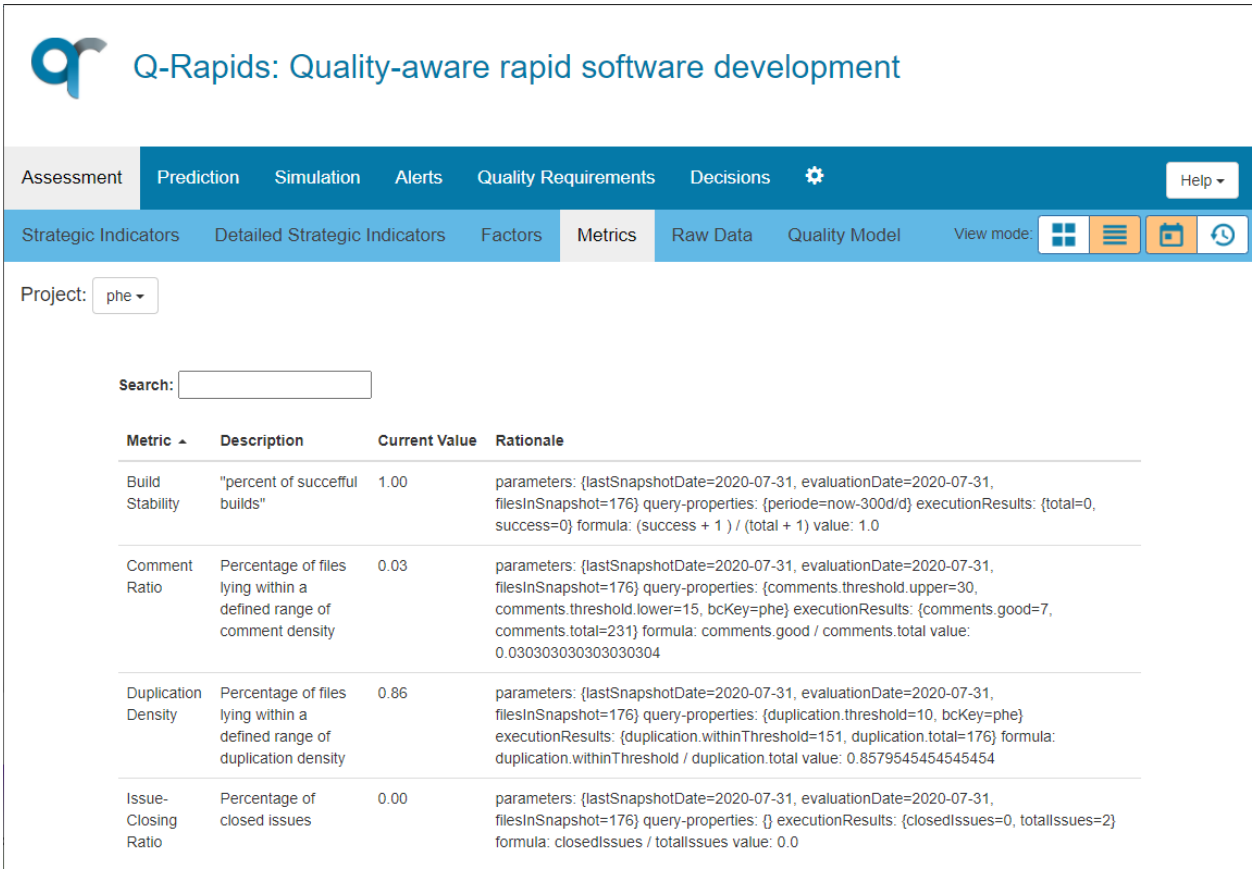


Fig. 3.b.IV. Description of metrics used.

c. Quality Use Case: Runtime Performance

Performance of software (more specifically the ability to perform specific tasks within a bounded time) is an example of a runtime (non-functional) quality. In this specific example we consider the performance of the different algorithms that make up the image processing pipeline of a professional production printer at CPP.

These image processing algorithms consume a lot of processing power and have to keep up with the printer speed in real time. The algorithms are developed in two stages. First, software is written to offline test new print modes and image processing algorithms in the image processing pipeline. In this stage, the images to be printed are processed offline and the resulting jetting sequences are fed into a printer or a printer simulation to see their effects in prints. In the second stage, the algorithms are moved to the inline software that performs the image processing in real time. This is the same code, but optimizations can be added. This is accomplished by the use of the language Halide that makes an explicit difference between the algorithm and the way it may be optimized: one can change the “execution schedule” without touching the algorithm. In both cases the performance of the different algorithms in the pipeline is logged. In the inline case this is part of the “functional logging” that continuously logs the status of machine parts, for example by means of sensor values.

In the DevOps cycle, a number of regression tests are performed each time a new build is made. With respect to image processing, this also includes making a number of virtual test prints with the embedded software connected to a software-in-the-loop (SIL) simulation of the printer.

In that case, the performance is as it would be in a real machine. The image processing related log files from the regression tests for each build are copied to a permanent storage area for further analysis.

Because the regression tests may run on different hardware platforms (they are executed on a test park of different hardware), performance results may depend upon the processor hardware of the test engine. A correction has to be applied to account for such differences.

Finally, the trends (for the consecutive builds) for the different print modes (the image pipeline and thus performance may differ per print mode family) are visualized via Jupyter (Python) notebooks, together with annotations for known changes in the software.

Fig. 3.c.I illustrates an example of the performance visualization for a specific print mode. Clearly visible is the performance degradation starting at build 7996. Note that some data only started to be available from build 7920.

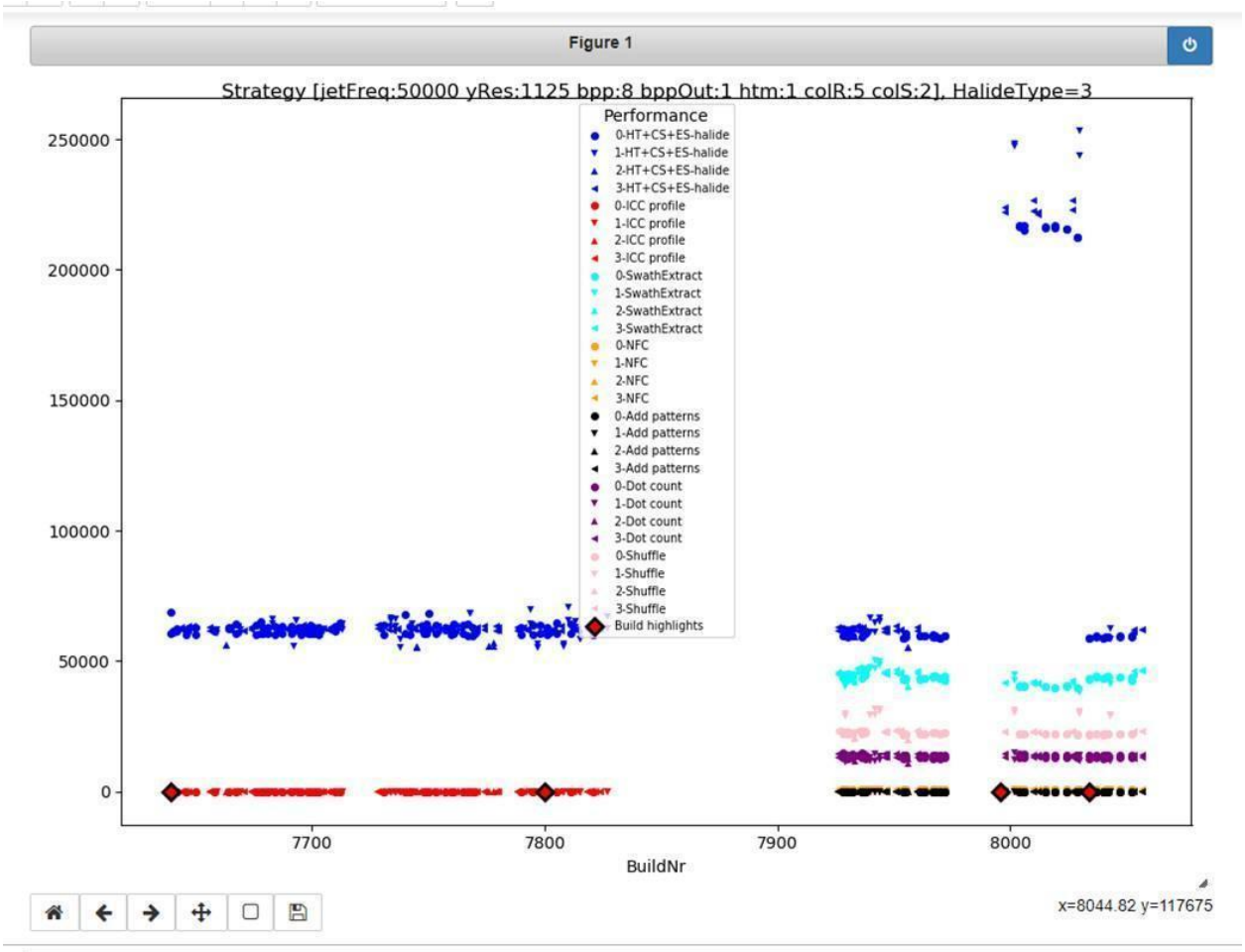


Fig. 3.c.I. Performance for a specific print mode.

The horizontal axis denotes the build number (based on last 4 digits), and the vertical axis the scaled execution time in microseconds. The execution time is scaled per hardware type to make execution times comparable.

Within the diagram a separate scatter plot is generated for each combination of [processing step, hardware type]. Each colour represents a different image processing step. Each marker shape represents a specific hardware type (i.e. the CPU type of the test host that ran the regression test).

The texts for the two build highlight markers that show the relation to the TFS (Team Foundation Server) item that has been created after the detection of this degradation are:

- 7992: TFS 300703(bug): Performance degradation for [jetFreq:50000 yRes:1125 bpp:8 bppOut:1 htm:1 colR:5 colS:2, HalideType=3]
- 8034: TFS 300703(bug): Fixed

d. Software as a Service (SaaS) Use Case

i. Background

Roadmap is a strategic planning tool that lays out the features planned for the software. It is a very common tool in the software development toolbox. On one hand, it facilitates the discussion on what features, technologies, etc the investments should be put on and on the other hand, it provides a big picture for the development team and other stakeholders.

There are a plethora of existing roadmapping tools available already, for example ProductPlan [9], Aha! [10], Wrike [11] and Roadmunk [12] to name just a few. These tools offer usually a Gantt chart view to what will be implemented in the product and when. While this is very useful, the tools fail to communicate why something is valuable and why it should be done at the certain moment. By gathering data from individuals and from various project management tools and processing, analysing and combining this data, we could create roadmap visualizations that are *based on the data* and are *living*: if the data changes, the roadmap adjusts accordingly to maximize the produced end user value based on the current situation.

ii. Goal

In international SaaS use case the goal is to create a roadmap visualization that facilitates the discussions on what needs to be done next from business perspective as well as from the technical point of view. To do that, we need to gather data on the roadmap level features, technical debt paybacks, improvements, etc., value them by stakeholders, create a synthesis of this data and visualize it in a way that helps to facilitate the discussion on the correct order of things in the roadmap.

From the resulting visualization, one should be able to tell when a certain customer will receive the new feature he or she has requested. The business owner should be able to tell based on the visualization what would be the optimal way to organize the roadmap to provide the best end user value according to the estimates of the stakeholders and how the currently selected roadmap differs from the optimal one. Sometimes, it is just necessary to deviate from the optimal roadmap, to allow payback of technical debt to keep the development speed on a good level or it just is more efficient to implement some features together, although the optimal way to produce end user value would indicate that only one of those features should be implemented now.

iii. Data model

In order to create such a tool, we have analysed what data are needed to produce useful visualizations of the roadmap for various stakeholders. Fig. 3.d.I illustrates the initial data model. In the centre, is the concept of task. Task in this context needs to be thought in a broad sense. It can be a feature request, technological improvement, need for architectural refactoring, etc. Tasks have dependencies to other tasks. The most obvious way a task is dependent on each other is that one task needs to be implemented before the second one can be realized. However, there might be other relationships between tasks, too. Task can be a composition of other tasks or have some other relationship.

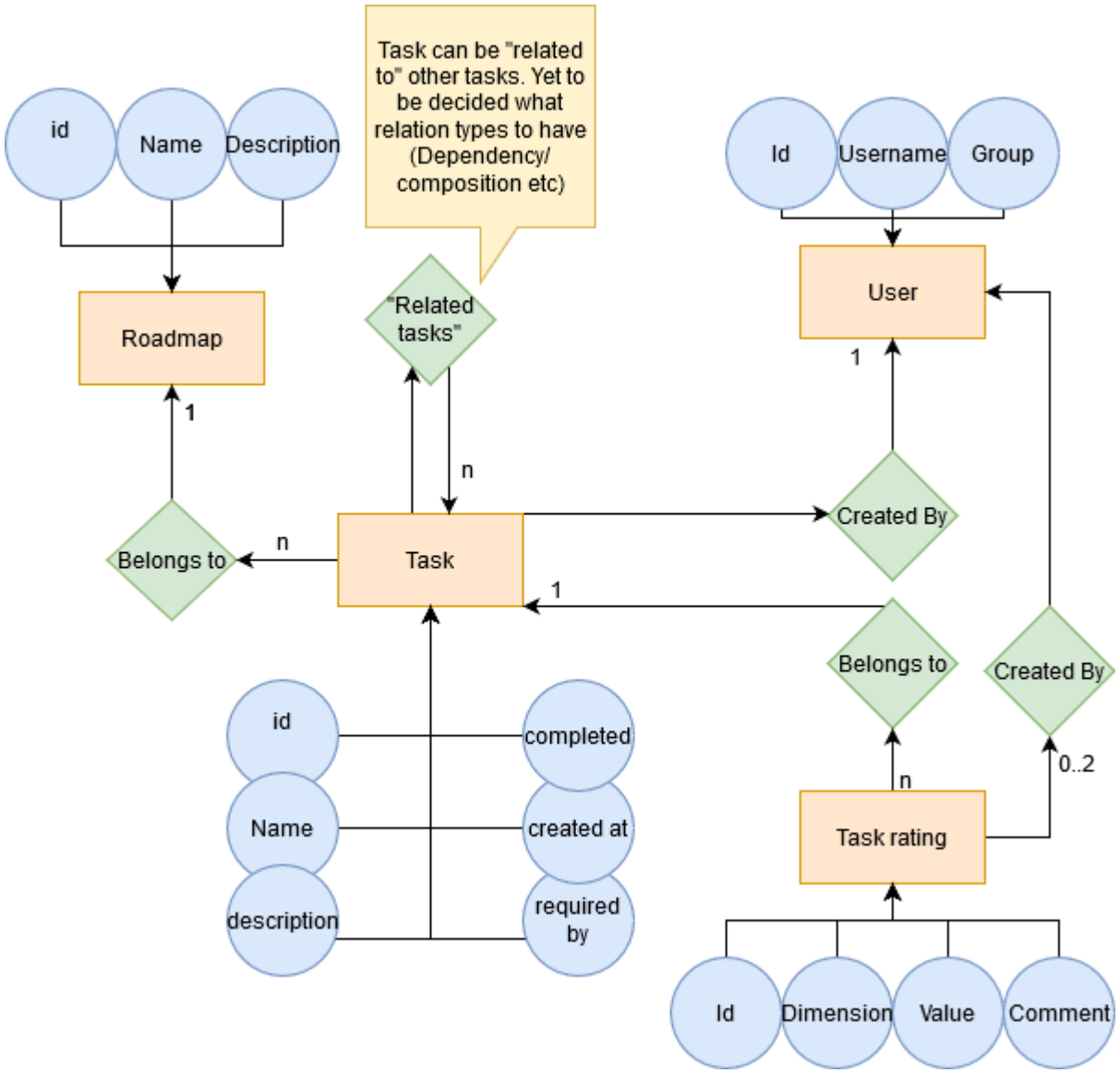


Fig. 3.d.I Initial data model for roadmapper tool based on the analysis. Note that roadmap segment concept is missing from this ER diagram.

We model different stakeholders as users of the system. These users have groups they belong to and the group defines their perspective to task ratings. Groups can be Product owner, developers, designers, account directors (or customer representatives), for example. Groups can be divided into two categories: business and development. These categories define from which point of view the user creates ratings for the task. Business category users create ratings on how important it is to get the feature done rapidly from a business perspective. Development category users rate tasks according to how hard they are to implement. From these two perspectives we can say that tasks with high business value and easy to implement could be done first. Or second, if we want to focus on high business value tasks that are hard to implement first.

Tasks belong to a roadmap and their ratings allow the system to organize them in an optimal order. The data model figure (Fig. 3.d.I) is missing a concept of the roadmap segment and it models a part of the roadmap. Segment is a group of tasks that needs to be or is beneficial to implement together.

In addition, user groups can have weights. This is particularly useful for account directors as it allows business owners to play around with the importance of accounts. We can weigh the ratings of certain accounts based on the perceived customer value. Then we can answer questions like "How would our roadmap change, if this certain customer turns out to be more important than we currently think?" or "How would our roadmap change, if the significance of this customer decreases?".

iv. Integrations

The description in the previous section focused on the minimum functionality for the roadmap visualization tool so that it is still useful. However, the real value of the approach lies in the integrations to other systems and data sources.

First and foremost is integration to project management systems such as JIRA, so that the tasks can be imported from and exported to the project management system. Export functionality is less critical. For this we need to implement a data fetcher and convert the data to fit the data model presented earlier. VISDOM data fetchers could do this.

In addition to JIRA, integrations to tools that provide code quality data could be useful. We could have a stakeholder view that would annotate the tasks on one dimension about technical debt, code smells and such data and order them according to urgency regarding these criteria.

v. Visualizations

From the data model we can visualize the roadmap data for example as depicted in Fig. 3.d.II and Fig. 3.d.III. In Fig. 3.d.II, a dashboard of potential roadmap visualization is shown. Heatmap shows an overview of business value and development effort ratings of the tasks. Planner plot visualizes the created user value of the currently selected order of the tasks and completion meter shows the percentage of completion of the current roadmap.

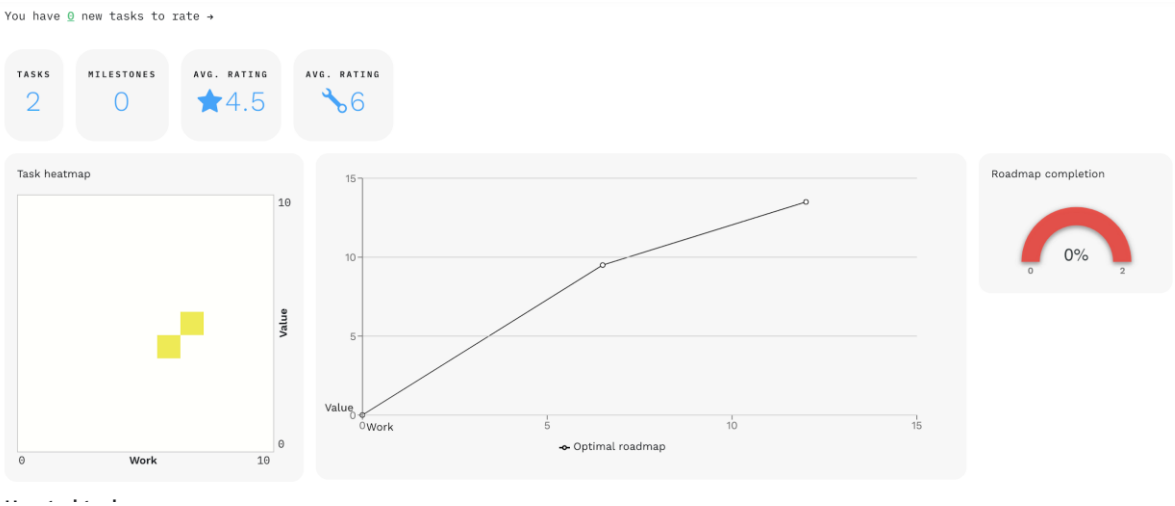


Fig. 3.d.II Possible dashboard of Roadmap visualization.

Fig. 3.d.III shows a project planner view where roadmap milestones are presented in a Gantt chart. This view helps in discussion when a certain feature or set of features is ready to ship.

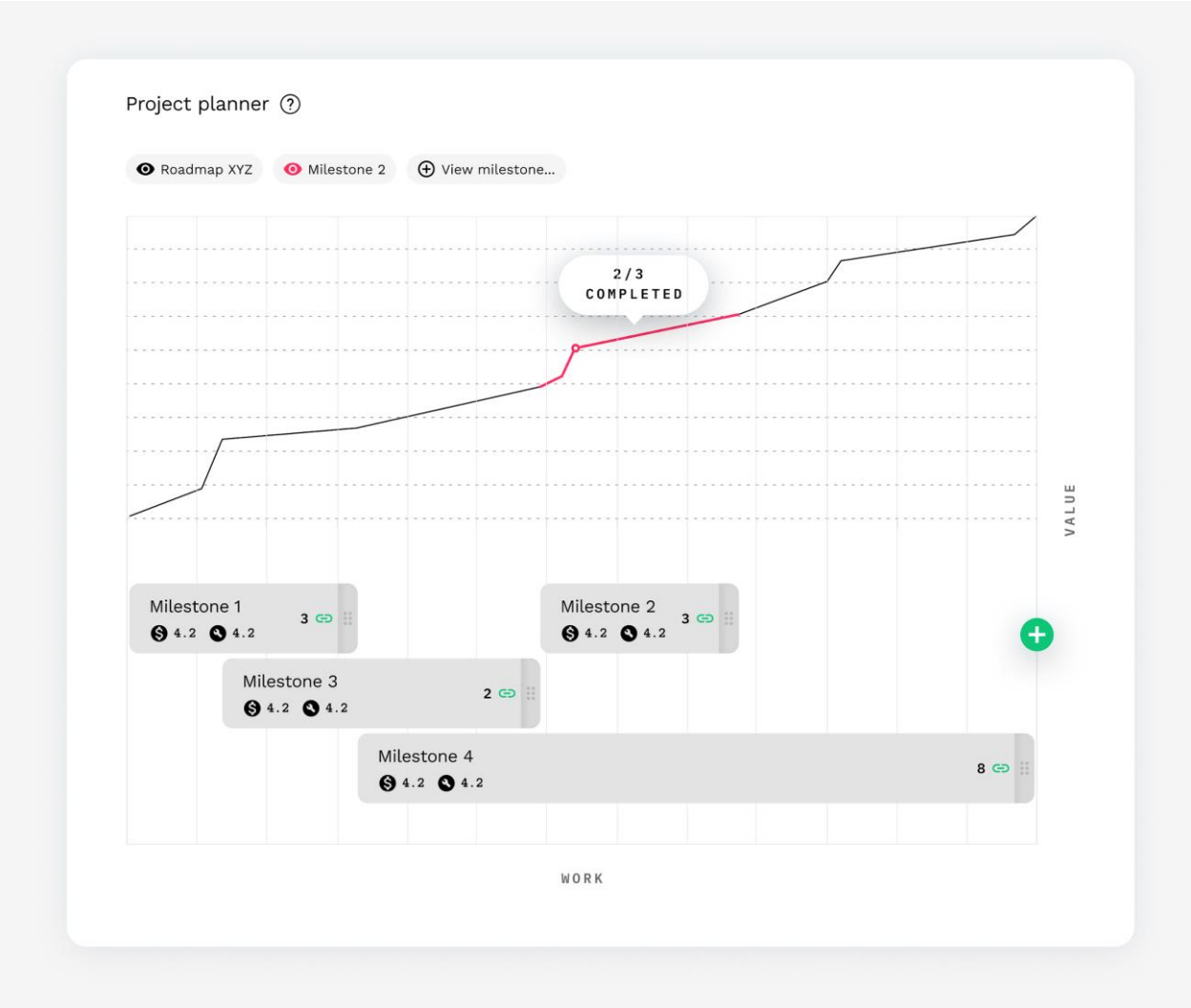


Fig. 3.d.III. Traditional Gantt chart view of the roadmap.

e. Teaching Use Case

i. Background

Visualizations have been used as part of software engineering teaching for decades. However, visualizations have been mainly targeted at the level of code, that is, to visualize the structure of software, rather than help in understanding the software development process. Our study on state-of-the-art solutions for visualizations in teaching [13] revealed only few approaches utilizing visualizations for process and project management. The majority of these approaches were based on Gantt charts, often used as part of a game-like simulator, providing students means to learn key concepts of project management, such as scheduling and resource allocation. However, none of the approaches we found used real data.

Another common way to visualize software process in teaching was to use burndown charts to track the progress of Scrum projects. These charts were usually quite simplified, as real-time approaches only concentrated on one-dimensional data, while more complicated approaches only provided analyses in a retrospective manner.

To summarize, the state-of-the-art study showed that 1) there are only few approaches using visualizations to teach software project and process management, 2) the existing approaches mainly use one view for students and teachers alike to show progress in a simplified manner, and 3) existing approaches use fairly standard charts and very limited real data.

ii. User study – needs for analysis

To identify visualization needs in teaching software engineering, we conducted interviews with 10 academics in the field of software engineering, teaching topics such as programming, software engineering processes, testing, software architecture, and software project management. While the discussions covered a wide range of software engineering topics and the teaching was aimed at both Bachelor's and Master's levels, the following critical needs could be found nearly unanimously in the interviews:

1) Need to see how students progress during the course

Teachers strongly brought up the need for visualizations on students' progress. They particularly desired a way to easily notice students who are starting to fall behind and are in need of extra support. Visualization of progress was also desired on various levels: on the level of an individual course and on the level of a whole study program. The teachers further felt a particular need for notifications on students who get stuck on automatically assessed assignments.

2) Need to see what topics are difficult to grasp

Along with being able to see how students progressed, teachers wanted visualizations on what are the topics that are difficult to grasp, thus creating bottlenecks for the course. On a more detailed level, there was a need to see the success/failure rates per topic, task or exercise.

3) Need to see how much effort students use for the given tasks

Finally, teachers needed to see how much effort students use. Effort estimations on coursework are based often on feedback given after courses, and they aren't very accurate. It would be highly valuable to see analyses on actual effort used, to better plan the coursework and to also communicate to students how much time and effort they should be prepared to invest in the course.
Additionally, the following needs were mentioned by some interviewees: visualizing work distribution within groups/between students, visualizing the pace at which groups start work on their projects (related to students' progress), visualization on code and how much code is rewritten during the course, monitoring the completion of different submissions.

iii. Requirements for analysis

1) Overview and background

Based on our user study, we focused on the three most critical needs and began to construct requirements for a visualization demo based on them. We treated each need as a use case and created scenarios around it to drive development of the visualization. Due to time constraints the current demo is built around one of these use cases: Monitoring/visualizing students' progress.
As teachers want to see how students are progressing, they need a visualization showing:

- The current "status" of each student
- Average status of completion
    - Simply viewing how one student is progressing may not show if they are falling behind. Comparing a student's progress to how the rest of the students are progressing will more easily help showing the ones in need of support.

- Expected status of completion
    - Based on history data from previous implementations, we can calculate how a student should be progressing at any given time in relation to a certain outcome (grade).
- Highlighting of students that are significantly below average

The "status" of a student is a multi-dimensional concept, calculated using several elements of data from various sources. In the current implementation for our teaching use case, "status" consists of the number of collected points and completed exercises per each course week and cumulatively up to each week, and the number of submissions and commits per each exercise. However, more elements can be added in the future. How a student progresses on a course is combination of the aforementioned elements, whose comprehensive investigation requires effective visualizations. Moreover, the most accurate information on progress can be obtained by comparing a student's status (or some elements of it) to what the status should be ("expected status") at a given point of time to ensure passing the course and achieving desired learning outcomes. That is, a student's status needs to be compared to the average student on the course, or to the expected progression based on history data to get a sense whether a student's status is "progressing nicely" or "falling behind". Visualizations, in turn, are used to show how students' statuses evolve over time and how different students' progress relate to each other.

Data mining and analysing available data to unearth a multi-dimensional value such as "status" is non-trivial. To get the most comprehensive picture of factors affecting a student's progress, we need to identify all relevant data sources and items and find relationships between them. While we are not using very complicated analytics in calculating values or processing the data (apart from calculations regarding history data), analytics are vital to form meaningful visualizations containing (analysed) data from a multitude of sources. Our use case demonstrates how manual analyses based on visualizations can provide much valuable information.

In our case, real live data currently comes from two data sources: GitLab code repositories and a MOOC-styled learning management system, Plussa. Teachers publish materials, weekly exercise assignments and give points using Plussa. Students have their own git-repositories that are hosted in Tampere University's instance of GitLab, and weekly programming exercises are submitted for automatic grading by submitting the repository url in Plussa, once an exercise is finished. Both GitLab and Plussa provide a simple, authorization-based RESTful API to the data saved in the systems. The data collected by Plussa includes course name, participants, exercises, submissions and collected points. GitLab API provides data about its users, repositories and commits.

However, only half of the course grade consists of collected exercise points, while the other half is made up of an exam that is taken and graded in a separate system. Thus, actual course grades are not yet available for use in the current visualization demo, as we cannot currently access data from the exam system. For this reason, course grades are projected from the collected exercise point data, and the estimates are used for testing the concepts of our analyses and visualizations that are based on history data. In the future, we plan to incorporate predictive analytics based in history data on grades from previous years in order to catch students early on who need more support to achieve the desired learning outcomes.

2) *Stakeholder interests*

There are three main stakeholder interests:

- **The teaching assistant (TA)** will want to check the status at certain intervals in real time during the course, and needs to see a visualization of the students' progress, and particularly some kind of highlighting of students who seem to be falling behind at that time.
- **The responsible teacher** might also check students' progress during the course, but he/she will be even more interested in examining the data and visualizations in retrospect – who were falling behind in the beginning / middle but were able to catch up, were they given support (i.e. did support work) or not (were the students just busy in the beginning or had some kind of epiphany and caught up on their own), and how accurately could predict difficulties in the end based on falling behind in the beginning. The responsible teacher would also be interested in larger trends and need visualizations on differences between study years.
- **A student** might be interested to see how he/she is progressing compared to the average student taking the course. A student can also see what kind of grade can be expected based on the current progression.

3) *Scenarios*

We constructed six scenarios to portray the different stakeholder interests. All scenarios depend on the following data, mined from code repositories (GitLab) and the task submission system (Plussa):

1. Number of completed tasks on a given week
2. Number of points gathered on a given week
3. Number of completed tasks cumulatively up to a given week
4. Number of points acquired cumulatively up to a given week
5. Number of commits per each submitted task per given week
6. Average number of commits per task up to a given week
7. Number of commits cumulatively up to a given week
8. History data from previous implementation on course
    a. Grades of students from previous implementation
    b. Number of tasks, points and commits from students at each given time point

Table 3.e.I lists the scenarios, the views they are linked to, and the data analyses required for each scenario. As listed, Scenario 6 additionally requires data from other courses in the study program. How the data is shown to different stakeholders and what kind of analyses are provided differs between scenarios and is presented in different views.

Table 3.e.I Scenarios for "Monitor student progress" requirement in Teaching use case.

	Scenario 1 Check current status	**Scenario 2 Receive notifications of those falling behind**	**Scenario 3 Check progress over time**	**Scenario 4 Compare progress with results**	**Scenario 5 Compare own status to others'**	**Scenario 6 Compare course progress to other courses**
View	Status view	Status view	Progress view	Results view	Student view	Course view

Stakeholders		Responsible teacher, TA	Responsible teacher, TA	Responsible teacher	Responsible teacher	Student	Responsible teacher
Real time data analyses	For individuals	Cumulative points from tasks, commits/task, number of submission attempts, and completed tasks	Finding those below certain threshold	Evolution of status over time, comparison of status to expected status over time	Correlation between grade and progress over time (in relation to expected)	Cumulative points from tasks, commits/task and number of completed tasks	Status as calculated from available data in relation to expected progress for all available courses
	For group	Average points, completed tasks and number of commits/tasks	Percentiles (showing how many are currently below selected threshold)	Evolution of average over time	Correlation between grade and progress for selection of individuals, evolution of group status over time	Average points, completed tasks and number of commits/tasks	Average statuses for all available courses
History data analyses		Relating grade data history and expected progress at a given time point	Thresholds for minimum expected	Evolution of progress thresholds over time, evolution of "expected level of completion" over time	Comparing grade distribution against previous implementations	Relating grade data history and expected progress at a given time point	Expected progress thresholds for all available courses

iv. Analyses provided by visualization demo

The current version of the visualization demo implements the status view and the progress view from the perspective of the responsible teacher or the teaching assistant. Both views provide visual analysis scenarios as described in Table 3.e.I. Both in the status view and the progress view, the main visualization component can be displayed in four different modes: the point mode, the exercise mode, the submission mode and the commit mode. Each of these modes represents a dimension of the student's status.

Fig. 3.e.I illustrates the status view in *point mode*, where each student is represented by a bar. Student bars are coloured according to how many points the student has received, and the bars are ordered by points accumulated over the course. The user may select to display a horizontal reference line for the average of collected points. Similarly, a vertical reference line for separating students that fall below a customizable threshold, is available. Fig. 3.e.I shows all gathered points from the first course week, as well the average accumulation of points (dotted horizontal line), and a threshold for students with a completion rate of less than 10 percent of maximum points (blue vertical line).

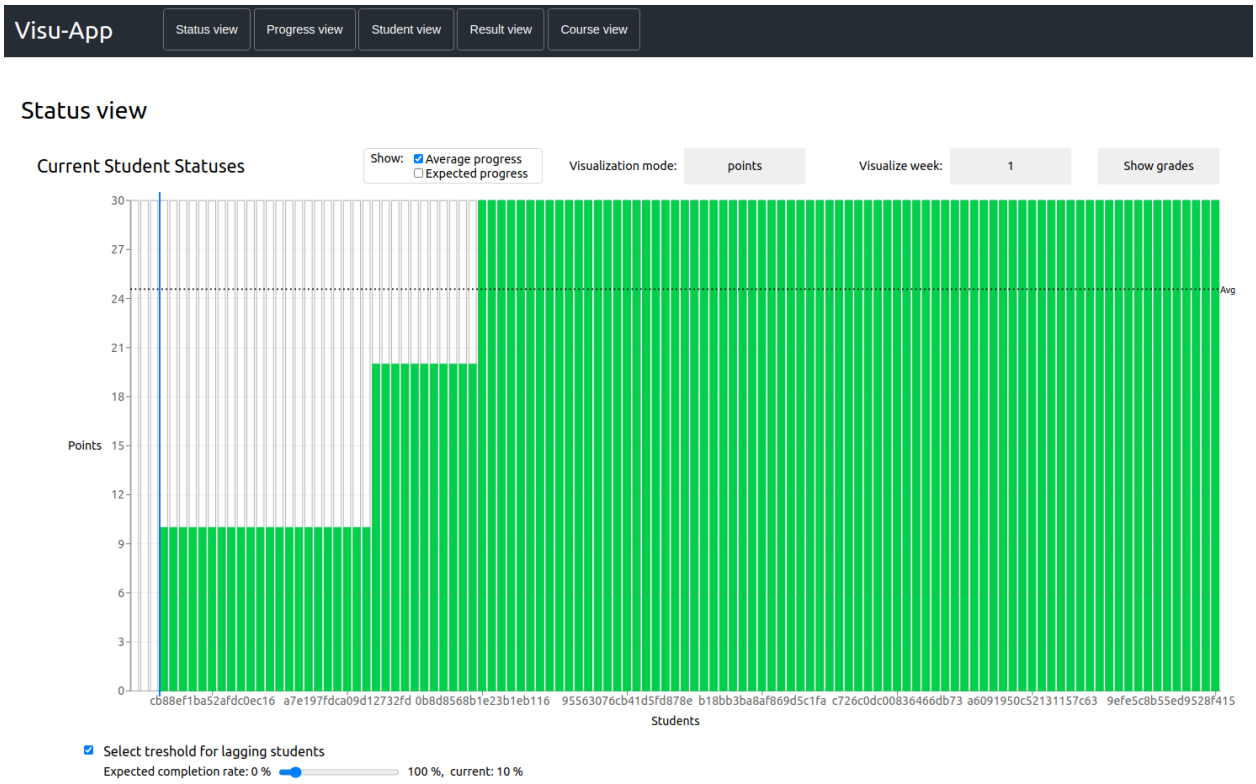


Fig. 3.e.I. A screen capture from the status view.

As the course progresses, the teacher can also use the point mode to inspect either any of the past or the ongoing course week to see how many points students receive each week, while also seeing the accumulation of received and missed points, coloured in red and dark green. This is illustrated in Fig. 3.e.II.

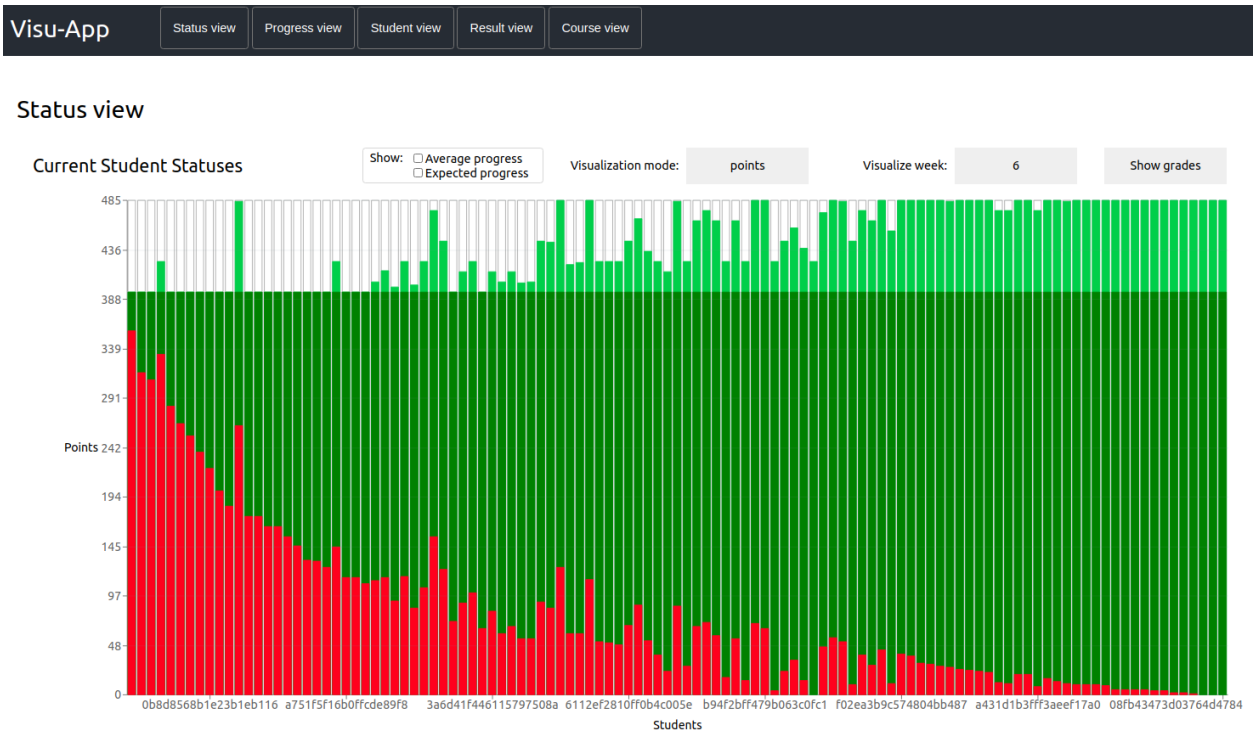


Fig. 3.e.II. Status view in point mode showing students' statuses on the sixth week of the ongoing course.

Fig. 3.e.III demonstrates the *submission mode* of the status view on the fourth course week. Exercise number 4 stands out visually, having relatively many white, dark green and red entries in comparison with the other exercises of the week. White entries symbolize a student not having made a submission to the exercise, red entries symbolize failed exercises and green shades passed exercises, a darker shade marking several re-submissions.

Fig. 3.e.III: Status view in the submission mode. Week number 4 seems to feature 5 exercises, with the fourth exercise visually standing out.

A quick look into the exercise details tells that the visually deviating exercise is a project, generally meant to be a broader, more difficult and comprehensive task than normal weekly exercises. Inspecting the same week in the commit mode, as shown in Fig. 3.e.IV, gives visual feedback to suspect the project to be more laborious than the other exercises on the same week, since the darker shades of blue signify a higher count of git commits made in the files submitted in the exercise.

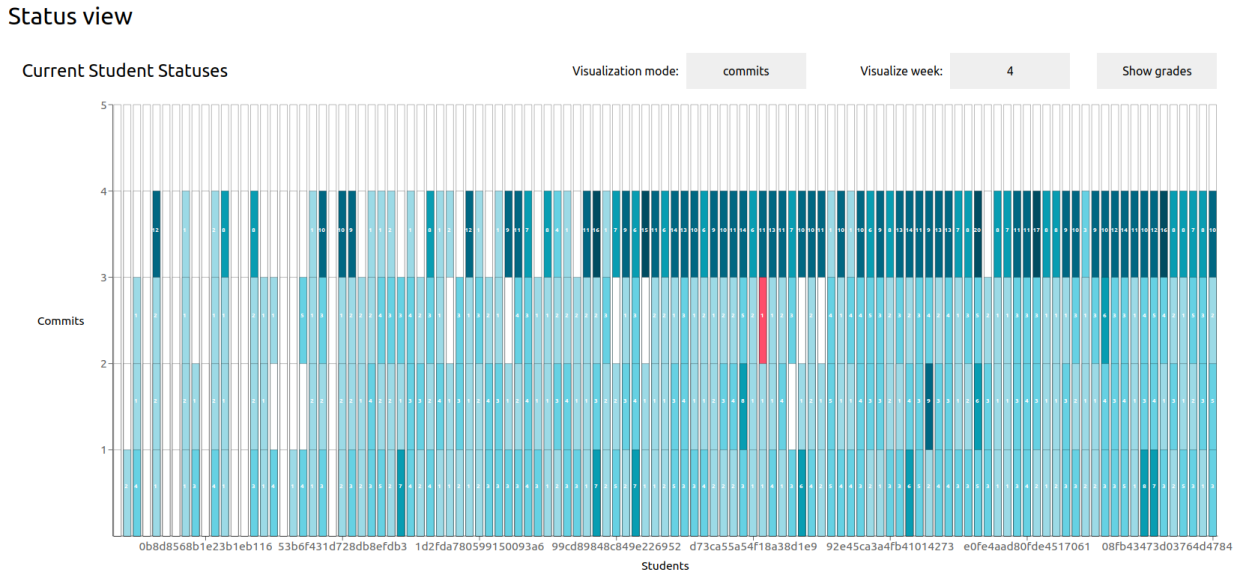


Fig. 3.e.IV: Status view in commit mode.

In contrast to the status view displaying student statuses one week at a time, the progress view displays the chosen aspect of a student's status for a given time span, demonstrated in Fig. 3.e.V. The view consists of two visualizations, the first of which showing how many points students received each week and the second one showing the accumulation of points over the course. The view allows the teacher to inspect the evolution of students' statuses on the course over time. The starting and ending points of the timeline are selectable via a slider, allowing a zoomable overview over the course.

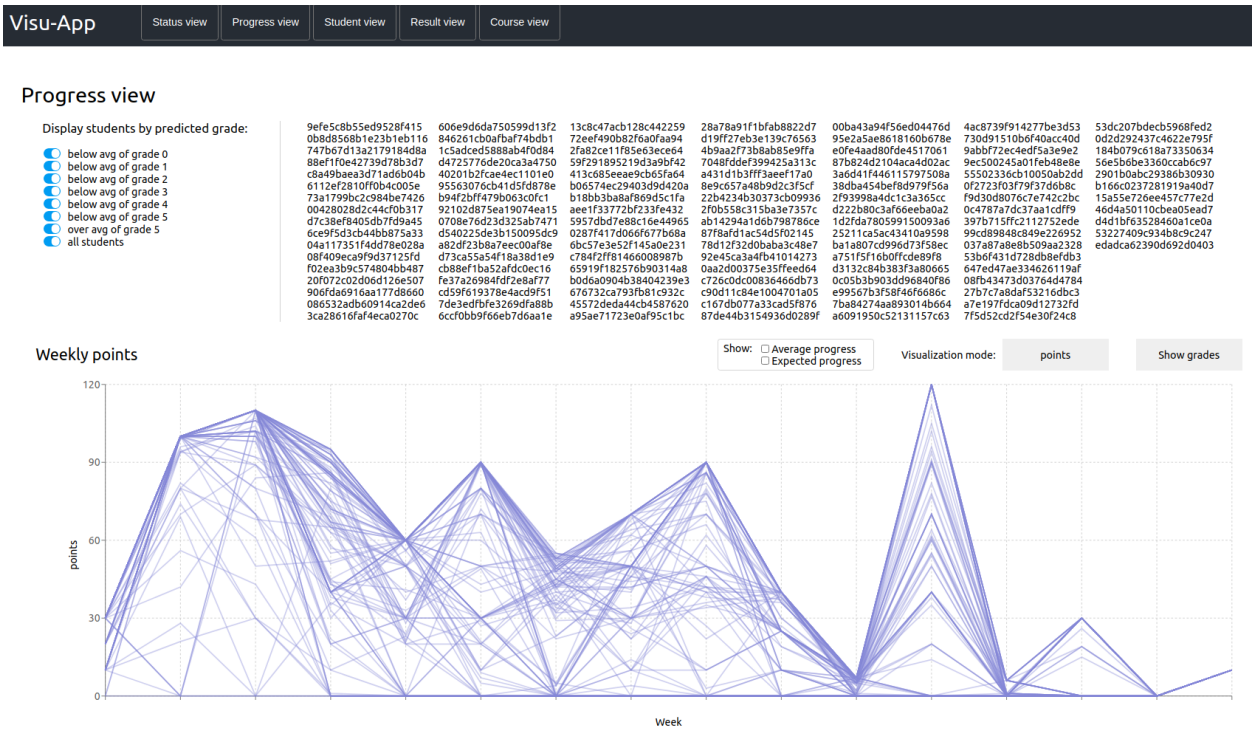


Fig. 3.e.V: The progress view in point mode, showing how many points each student has received during the course.

The teacher may select a subset of students and reference lines to inspect how the status of a certain group of students evolves in relation to other students or the average student, and which grade(s) the selected students may expect, based on history data. For example, Fig. 3.e.VI shows a student status line in purple, the average student in grey and six green reference lines showing the expected progress on the course based on history data.

The reference lines are calculated from the average of collected points among students that have received the same grade on the previous course implementation. The approach aims at providing an estimation on how a status evolves when it is about to result in a certain grade. As the expected lines reflect averages from history data, falling below or rising above them during the current course implementation does not guarantee any certain grade. However, some characteristics can be attributed to different student groups based on their progress profiles.

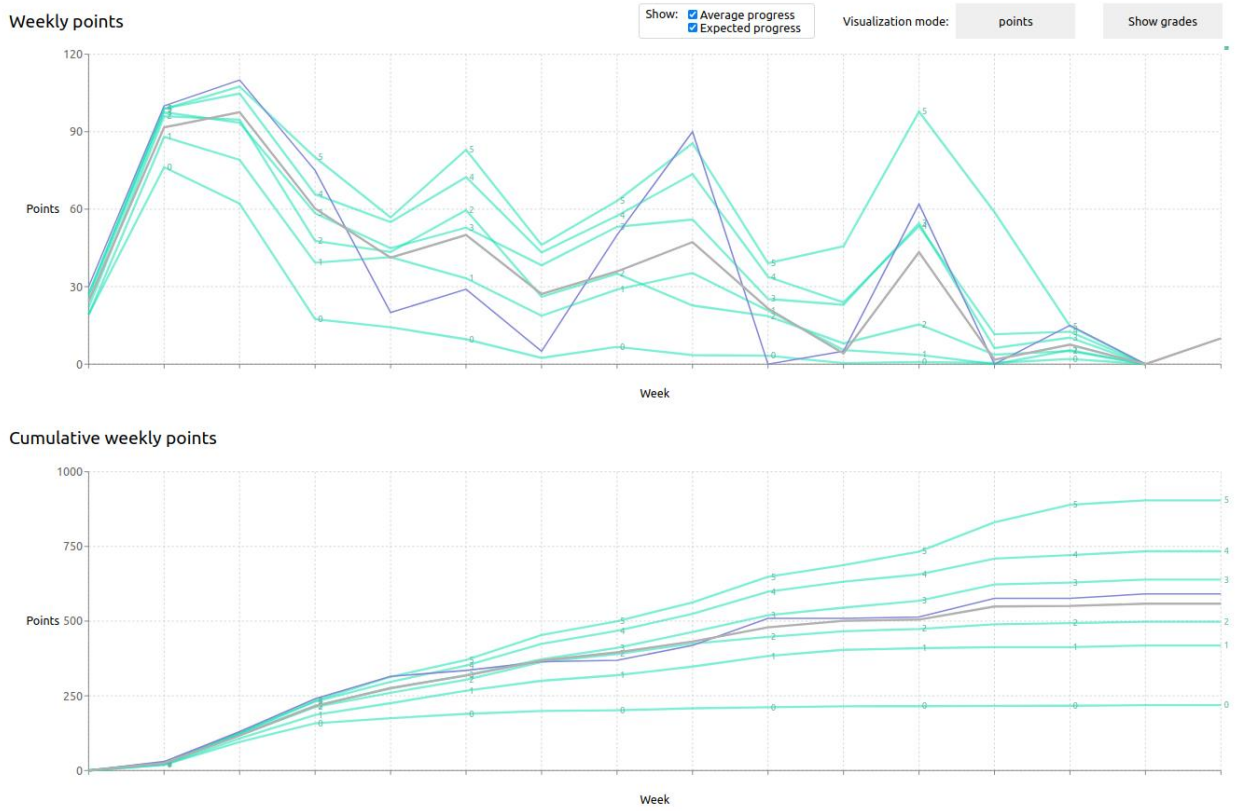


Fig. 3.e.VI: Inspecting the evolution of a student statuses in progress view. Cumulative weekly points suggest that based on history data, the student is expected to receive grade 2.

Some of the grade group characteristics can be found by inspecting Fig. 3.e.VII and Fig. 3.e.VIII. Fig. 3.e.VII shows a selection of students, whose accumulated points reside between the expected lines for grade 2 or 3. Fig. 3.e.VIII shows a selection of students that are likely to receive grade 5. Comparison of the two groups shows visible difference in the amount of variation on *when* and *how much* points are received. Students likely to receive grade 2 or 3 may have started working on the course later and even skipped exercises altogether on some week, whereas students going for the top grade are receiving high points throughout the course.

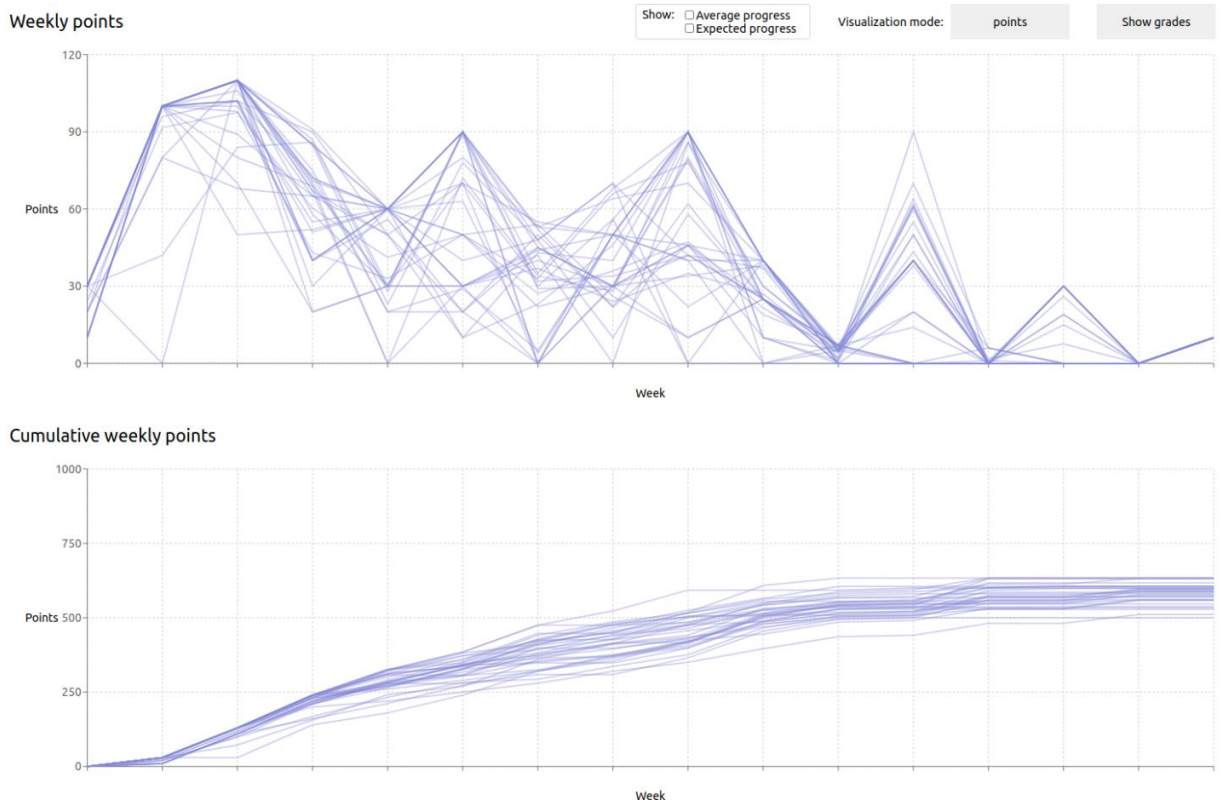


Fig. 3.e.VII: Selected student group consists of students that have received total of points that leaves them between the average point sum of grade 2 and 3.

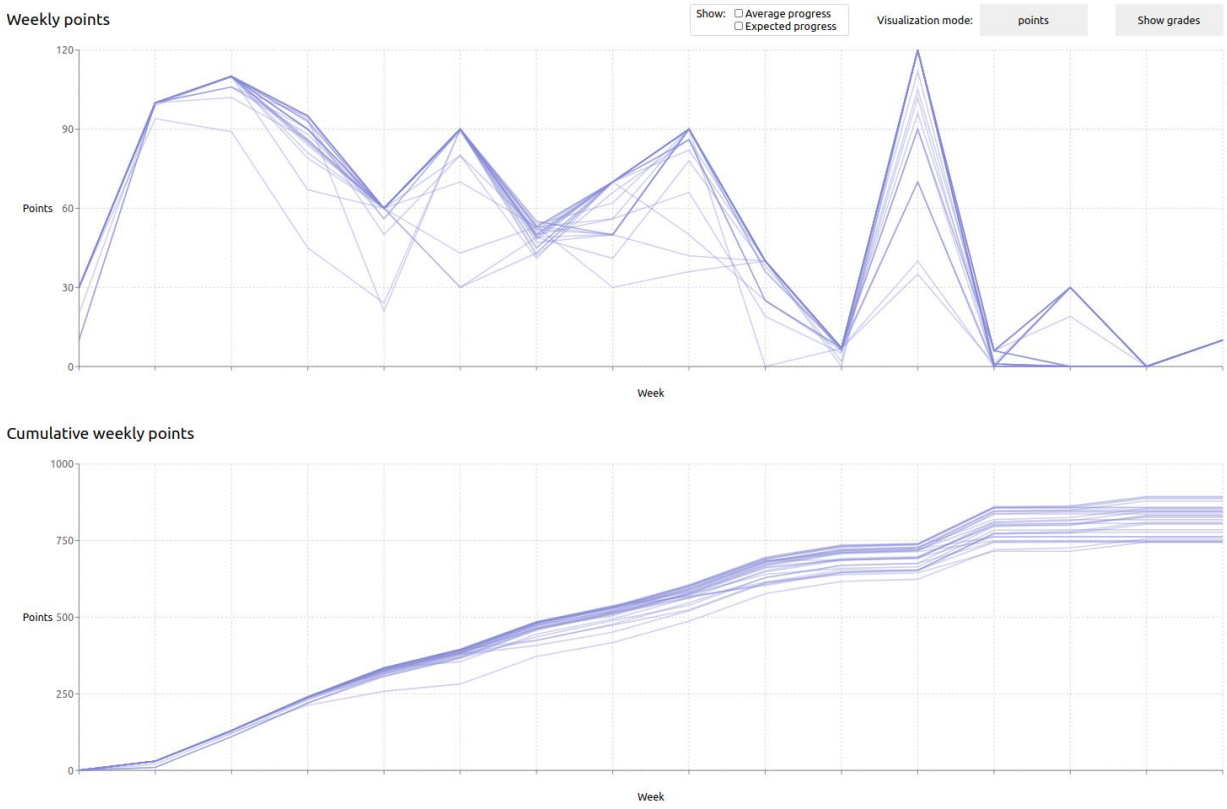


Fig. 3.e.VIII: Progress of students that have accumulated enough points to likely receive grade 5.

Fig. 3.e.IX shows the average commit counts for each grade, which provides a few interesting notions. The cumulative commits chart shows that students having received grade 0 seem to have stopped working on the course around the fourth course week on average, since until then, the zero reference line seems to remain close to the rest of the reference lines, and after that not to rise at all. Perhaps coincidentally, the first project exercise of the course is on the fourth course week, as noted previously.

Furthermore, there seems to be little difference in the commit count profiles of students that have received grade 1 or 2, since the average commit counts remain very close to each other for the whole course. The same phenomenon seems to exist between students that have received grades 3 or 4. In contrast, the grade 5 students are separated from the rest by keeping a high commit count up until the end of the course.

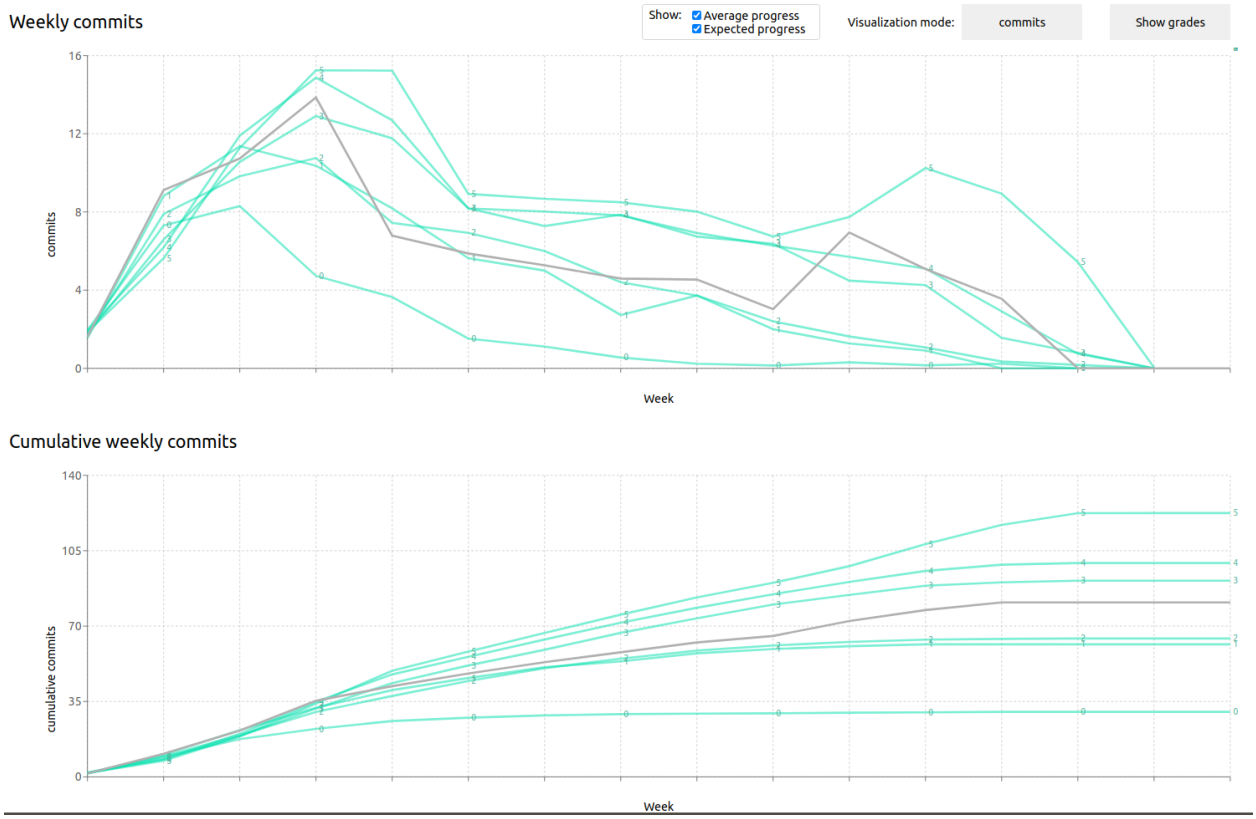


Fig. 3.e.IX: Grade-wise average commit counts from the previous course implementation.

In addition, a certain kind of a quick workload estimate for the course can be made from the cumulative commit counts shown in Fig. 3.e.IX. By the end of the course, students having merely passed the course, have made about half the number of commits made by the students that received grade 5. This trend seems to exist throughout the course, according to the weekly commits graph in the same figure. Similarly, the average student from the current course implementation (shown in grey) seems to remain between the average commit accumulation of students that have received grade 2 or 3. A loose interpretation of the fact could be that the average student seems to get about average grades.

Despite being still work-in-progress, the current demo provides already some visual insights into the visualization needs recognized in the teacher interviews. Provided insights include views into

the current student status and average status of completion, and these are available in both implemented views, as well as the expected status of completion and highlighting students falling behind with the course work. While visualizations do not provide definitive answers, they highlight interesting patterns in the data. When patterns are recognized, they can be formed into preliminary hypotheses, which can then be confirmed or refuted by hypothesis testing methods. Next steps in the development work include implementing views for students and teaching assistants, as well as novel type of visualizations for student status.

4. Conclusions

This deliverable first made a link between data analysis (the objective of Task 2.3) and data fetching (the objective of Task 2.1) as well as data modelling (the objective of Task 2.2). To achieve that, it provided examples of acquiring data from a number of different sources, such as issue trackers (Jira, GitLab), commit messages (Git), quality metrics (SonarQube), and testing environments (Jenkins). It also demonstrated two data models: one for Technical Debt and a second for Product Quality, both being currently developed in the context of the Quality Use Case. The core part of the deliverable, subsequently provided five extensive examples of data analysis organized according to the three project use cases (Quality, SaaS, Teaching). Especially the Quality use case was further divided into technical debt, runtime performance and product quality. The range of analysis examples demonstrates the breadth of data that can be analysed and visualized in a DevOps environment. It also highlights the customization of analysis that can be performed in order to serve different goals and address different stakeholder concerns.

As a next step, we are working towards the integration of the different tools into the VISDOM toolchain and subsequently the visualization of the analysis results in the envisioned dashboards. While the five analysis examples that were presented in this deliverable were developed independently of each other, there are opportunities of integrating the analysis across the corresponding tools. For example, the data and corresponding analysis from the quality use case (especially regarding Technical Debt and Product Quality) can be integrated in both the SaaS use case and the Teaching use case. Within WP3 we will explore such synergies between the use cases, and where possible implement integration of the tools developed for the individual use cases. Finally, the goal of this deliverable was to present some examples of analysis, while we plan to develop further analysis tools during the rest of the project. For example in the case of the Technical Debt part of the Quality use case, we are currently integrating a number of different data sources (issues, pull requests, commit messages, source code comments and code reviews) to automatically detect self-admitted technical debt using machine learning techniques.

5. References

1. Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” J. of Syst. and Software, vol. 101, no. C, pp. 193–220, Mar. 2015.
2. E.d.S. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical debt,” in 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD). IEEE, 2015, pp. 9–15.
3. E.d.S. Maldonado,R. Abdalkareem,E. Shihab,and A. Serebrenik, “An empirical study on the removal of self-admitted technical debt,” in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2017, pp. 238–248.
4. Nicolli SR Alves, Leilane F Ribeiro, Vivyane Caires, Thiago S Mendes, and Rodrigo O Spínola. 2014. Towards an ontology of terms on technical debt. In 2014 Sixth International Workshop on Managing Technical Debt. IEEE, 1–7.
5. S. Martínez-Fernández, A. M. Vollmer, A. Jedlitschka, X. Franch, L. López, P. Ram, P. Rodríguez, S. Aaramaa, A. Bagnato, M. Choras, and J. Partanen, “Continuously assessing and improving software quality with software analytics tools: a case study,” IEEE Access 2019, vol. 7, pp. 68219 - 68239, May 2019.
6. H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” The annals of mathematical statistics, pp. 50–60, 1947.
7. R. J. Grissom and J. J. Kim, Effect sizes for research: A broad practical approach. Lawrence Erlbaum Associates Publishers, 2005.
8. L. Lopez, S. Martínez-Fernández, C. Gómez, M. Choras, R. Kozik, L. Guzmán, A. M. Vollmer, X. Franch, and A. Jedlitschka, “Q-Rapids Tool Prototype: Supporting Decision-Makers in Managing Quality in Rapid Software Development (Demo),” In: Mendling J., Mouratidis H. (eds) Information Systems in the Big Data Era. CAiSE 2018. Lecture Notes in Business Information Processing, vol 317, pp. 200-208, June 2018. Springer, Cham. https://doi.org/10.1007/978-3-319-92901-9_17
9. Product Plan, website, https://www.productplan.com/, visited 14.9.2020
10. Aha! Roadmap tool, website https://www.aha.io/, visited 14.9.2020
11. Wrike, website, https://www.wrike.com/, visited 14.9.2020
12. Roadmunk, website, https://roadmunk.com/, visited 14.9.2020
13. VISDOM D1.1.1 Public state of the art document https://itea3.org/project/workpackage/document/download/6093/D1.1.1_Public%20state%20of%20the%20art%20document.pdf, visited 18.9.2020