28.01.2021

D2.3.2 Documented definitions of the metrics

Contents

REVISION HISTORY			
Version	**Date**	**Author/ Organization**	**Modifications**
0.1	01/10/2020	Paris Avgeriou (RUG)	Drafted outline and executive summary
0.11	09/11/2020	Outi Sievi-Korte and Vivian Lunnikivi (TAU)	First draft of section 2.e
0.2	30/11/2020	Lou Somers (CPP)	First draft of section 2.c
0.3	16/12/2020	Paris Avgeriou and Yikun Li (RUG)	First draft of section 2.a
0.4	21/12/2020	Outi Sievi-Korte (TAU) and Lidia López (UPC)	Updated section 2.e and first draft of 2.b
0.5	21/12/2020	Veli-Pekka Eloranta (Vincit)	Updated section 2.d
0.6	22/12/2020	Paris Avgeriou and Yikun Li (RUG)	Updated section 2.a and overall deliverable
0.7	13/01/2021	Paris Avgeriou (RUG)	Worked out review from TUT
0.8	13/01/2021	Lou Sommers (Canon)	Worked out review from TUT
0.9	13/01/2021		Updated section 2.b (background and data)
1.0	25/01/2021	Paris Avgeriou and Yikun Li (RUG), Lou Sommers (Canon), Veli-Pekka Eloranta (Vincit)	Updates based on reviews
	28/01/2021	Lidia López (Experis)	Updated section 2.b (new metrics)

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 data (Task 2.1), and model data (Task 2.2) from software project repositories and product artifacts. Task 2.3 results in two deliverables: D2.3.1 which focuses on examples of data analysis, and D2.3.2 (the current deliverable) which focuses on data metrics.

In WP1 we have defined three different use cases, where each one focuses on a different aspect or application domain (software quality, Software as a Service – SaaS, teaching), and hence different tools and corresponding data. Therefore, in this deliverable, we will define for each use case, the appropriate metrics used in the analysis approaches. Accordingly, this deliverable is organized in line with the three use cases. The first use case that focuses on Software Quality is more comprehensive than the other two as it comprises three different aspects: technical debt, runtime performance and product quality. Each one of these three aspects is elaborated within a different subsection; this results in a total of five sets of metrics (technical debt, runtime performance, product quality, SaaS, teaching). We also note that some of the use cases are very specific focusing on the actual products and services of the project partners. The runtime performance use case concerns the printers of Canon, the SaaS use case focuses on the service of product road-mapping offered by Vincit, while the teaching case is focused on programming courses using a specific learning management system at University of Tampere.

The five sets of metrics are defined and presented independently in this deliverable. However, there are 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 the object of study within the visualization dashboards and are discussed here as future work within WP3 (see Section 3).

The rest of this deliverable is structured as follows. Section 2 elaborates on the definitions of the selected metrics. As aforementioned, the definitions are categorized according to the three use cases of the VISDOM project. Five sets of definitions are discussed in detail: three for the quality use case and two for the SaaS and teaching use cases. Finally, Section 3 concludes this deliverable and briefly outlines current and future work.

2. Definitions of metrics per use case

The next sub-sections elaborate on definitions of the metrics adopted in each use case (quality, Software as a Service – SaaS, teaching). As aforementioned, the quality use case is further decomposed into three aspects: Technical Debt, Product Quality, and Runtime Performance (respectively subsections 2.a., 2.b and 2.c). The list of all metrics organized per use case (including the three sub-use cases of Quality) is summarized in Table I.

TABLE I
METRICS FROM ALL USE CASES.

Use case	Metric
Quality - technical debt	Number of issues per TD type
	Number of issues per TD indicator
	Number of TD issues across time (evolution)
	Number of TD issues in temporal phases (before creating issues, during code review, after the patch)
	Percentage of repaid TD
	Percentage of who repays TD
	Repayment time (per TD item)
Quality - product quality	Percentage of closed issues
	Percentage of closed issues that are prioritised as “critical”
	Percentage of issues closed with a maximum of +-10% of deviation. This metric has been defined separately for each of the three development phases (Definition, Development, and Testing)
	Percentage of issues closed with a maximum of +-10% of deviation.
	Percentage of issues closed with a schedule deviation between +-10 and +-20%
	Percentage of issues closed with a schedule deviation higher than +-20%
	Percentage of closed bugs in the product backlog
	Percentage of critical bugs closed with respect to the total number of bugs
	Percentage of files lying within a defined range of comment density
	Percentage of files lying within a defined range of duplication density
	Percentage of successful builds in a certain period
Quality - runtime performance	MPBE (Mean Prints Between Errors)
	Amount of software errors per square meter printed
	Amount of open and solved P1/P2/P3/P4 problems
	Number of automatic tests
	Software quality items resolved
	Overall CPU load (usage) during sustained operation
	Average and peak elapsed time for each image processing step
	Trend in average and peak elapsed times during regression tests
Software as a Service	Complexity of features to be developed
	Development time (estimate)
	Development time (realized)
	Costs
	Value for the customer
	Value of the customer
	Overall value of the feature for the product

	Value for developers
Teaching	Current status of the student (points, commits, and exercises)
	Average status of the student (points, commits, and exercises)
	Expected status of the student (points, commits, and exercises)
	Time used for each exercise (sum of used time per commits made for said exercise)
	Time used each week (sum of time used for exercises on a given week)
	Average, median, minimum and maximum hours spent per exercise
	Average, median, minimum and maximum hours spent per week
	Relation between used effort and number of completed tasks
	Relation between used effort and expected grade
	The top percentage of tasks by 1) least effort used, and 2) highest level of completion
	The bottom percentage of tasks by 1) most effort used, and 2) lowest level of completion
	The average and median effort for 1) all exercises on the course, 2) all exercises for a specific week, and 3) all exercises on a given period
	The percentage of students completing 1) each exercise on the course, 2) exercises on a specific week, 3) exercises on a given period
	Percentage of code that is rewritten between commits
	Metrics gained from inspecting code, such as given by SonarQube
	Metrics related to style guidelines
	Percentage of code commits made by each member of the group
	Issues opened/answered by each member of the group
	New lines of code per commit by member of group
	Contribution to README etc. documentation per person
	Metrics related to the pace of work

a. Quality Use Case: Technical Debt

a.1 Background

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 Deliverable D2.3.1, we had defined the Data Model used in this use case, comprised of a number of types, and their corresponding indicators, as listed in Table II. These types and indicators are used in the set of metrics listed below.

TABLE II
DEFINITIONS OF INDICATORS OF DIFFERENT TYPES OF TECHNICAL DEBT IN ISSUE TRACKERS.

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 organized.

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 synchronization problems or efficiency problems.
	Slow algorithm	A non-optimal algorithm is utilized 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.
Requirement debt	Requirements 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.

a.2 Data

Data from issues are collected from open source Java projects that are of high quality and supported by mature communities. To select projects pertinent to our 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. This allows us to compare the results between our study and other self-admitted technical debt studies.

Furthermore, we select projects that use Git as a source code repository and JIRA[1] as an issue tracker. The number of Java files and SLOC are calculated using the LOC tool[2]. The number of contributors is obtained from GitHub. We used the JIRA Python package to extract all issues from the online server and stored them in a local database; then we counted the number of issues.

a.3 Metrics

We have selected a number of metrics based on their usefulness for software practitioners [4]. For each metric, we provide: a name, a definition, an example that helps to understand the definition,

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

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

as well as its potential usage by software engineers. The examples are derived from two Apache projects, namely Hadoop[3] and Camel[4], and they refer to specific issues of those projects in JIRA.

- *Number of issues per TD type*

Definition: for each type of TD (the list of types of TD are shown in Table II), the total number of issues containing this type of SATD is counted.
Example: in Hadoop-9763, the reported TD is *test debt*, because the developer commented 'still need to test it.' For Hadoop, we have 20 instances of test debt and 6 instances of design debt.
Usage: this metric helps developers gain an overall understanding of the proportion of different types of TD in the project. If a certain type of TD is significantly accumulated, it can be prioritized for repayment.

- *Number of issues per TD indicator*

Definition: for each indicator of TD (the list of indicators of TD are shown in Table II), the total number of issues containing this indicator is counted.
Example: in Hadoop-8288, there is a report of redundant code: 'they are not being used any more'; thus, the indicator of this TD is *dead code*. We found 3 instances of dead code in Hadoop.
Usage: this metric helps developers gain an overall understanding of the indicators that cause TD in the project. When the TD of specific indicators exceeds a specific threshold, the development team needs to discuss about this indicator, and what measures can be further taken to address it.

- *Number of TD issues across time (evolution)*

Definition: during software development, some TD items are introduced while others get resolved. This metric counts the number of issues containing SATD across time and is usually visualized as a time series (e.g. number of issues on the y-axis and time on the x-axis).
Example: under the pressure of delivery, developers may choose to incur more TD, leading to the accumulation of TD. For instance, in Camel, we had identified 37 TD instances on the date of 30 Oct 2012, and this number rose to 51 TD instances on the date of 1 Feb 2015.
Usage: keeping track of how the number of TD issues fluctuates along time is crucial for TD management. Developers can identify spikes in the evolution, where significant amounts of TD were accumulated. They can also see the overall trend, i.e. how fast TD is increasing in their system.

- *Number of TD issues in temporal phases*

Definition: TD can be identified and reported in three different time points: 1- before creating issues, 2- during code review, 3- after the patch is made. The numbers of SATD declared in each of these three periods are counted.
Example: in Camel-903, documentation debt was reported after committing the patch - 'todo: need to document it in wiki.' In Hadoop-12923, architecture debt existed in the system and later on was reported in the issue description - 'some code is used only by tests. let's relocate them'. For Camel, we found 27, 13, and 11 instances of TD reported before creating issues, during code review, and after committing the patch respectively.

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

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

Usage: this metric helps developers gain an overall understanding of the proportion of TD reported in different temporal phases. This is particularly important for the code review process. If more and more TD is reported after submitting patches rather than during the code review, then the code review process should be revisited and potentially improved. Similarly if only few TD items are reported in the issue description, it could be a problem with the organization culture that does not encourage good code quality assessment.

- *Percentage of repaid TD*

Definition: after TD is reported in issue trackers, some of the TD is resolved and reported in issues. The numbers of repaid and unrepaid TD items are counted, and then the percentage of repaid TD is calculated (by dividing the former by the total).
Example: in Camel-231, the TD was first reported in the issue summary - 'broken link on wiki page.' Then the developer fixed it and reported the repayment - 'fix applied.' We found that 71.3% and 72.5% of TD were repaid in Hadoop and Camel respectively.
Usage: this metric provides developers the proportion of repaid TD compared to unrepaid TD. If more and more TD is reported but not paid off, developers should be urged to spend more effort on TD repayment.

- *Percentage of who repays TD*

Definition: TD can be paid off by those who create it, those who identify it, or those who participate in resolving it. The numbers of TD instances paid by these different types of developers are counted. Then the percentage of TD paid by each type is calculated.
Example: in Camel-201, test debt was reported by its creators, because the patch creator did not know where to put the tests - 'there are no XQuery specific tests (mainly because of not knowing where to put them).' In Hadoop-8124, the TD was reported by its identifier, because the code reviewer noticed a deprecated function was used in the patch - 'the syncable.sync() was deprecated in 0.21, we should remove it.' For Camel, we have 12 instances of TD repaid by its creators, 19 instances of TD repaid by its identifiers, and 6 instances of TD repaid by others.
Usage: this metric provides developers the proportion of TD paid by different types of people. Debt creators are the most knowledgeable to pay back debt and can do it more efficiently. Thus, if most of TD is repaid by others than those that incurred the debt in the first place, TD repayment may be neither effective nor efficient. Depending on the values of the metric, a development team may assign more repayment tasks to those that incurred the debt.

- *Repayment time (per TD item)*

Definition: the time spent on repaying a TD item is calculated by subtracting the time it was reported from the time it was resolved.
Example: in Camel-1320, after 16 hours of spotting the a TD item - 'we need to add it to the wiki page,' the documentation debt was resolved. Thus, the repayment time of this TD item is 16 hours. The average TD repayment time for Camel is 633.1 hours.
Usage: this metric indicates how quickly or slowly different types of TD are resolved. If it takes a longer time to repay a certain type of TD, this type of TD can be given a higher priority for repayment.

b. Quality Use Case: Product Quality

b.1 Background

To identify visualization needs in the context of product quality monitoring for Experis projects, we conducted a workshop including project manager, product owners, system analyst, developers and server technician. During the workshop, using Goal-Question-Metric (GQM) paradigm [3], we identified a set of indicators to be included in the demonstration. Project manager (responsible for both products) was seeing the progress and efficiency of the development and the quality of the final product. The quality of the product depends on the product, for the development team they are interested in the quality of the product in terms of bugs and for the operations team in terms of availability and performance. GQM paradigm allowed us to define the set of appropriate metrics to be able to assess the identified indicators.

b.2 Data

Data produced by the DevOps team in Experis were provided by different kinds of tools. GitLab is the tool used to source code and project management, metrics defined for measuring product quality are mainly using data related to project management tasks, this data is used to assess the project evolution (issues, planned/real effort ...) and quality of service (bugs). Jenkins provides support to automated testing and continuous integration (tests and build information). Additionally, the development team uses SonarQube to perform static code analysis to improve the quality of the source code. In order to support operations teams, some runtime data will be used to assess the quality of service, including availability and performance.
This data can be complemented with data described in Section 2a: data about technical debt could be used to complement the code analysis provided by SonarQube.

b.3 Metrics

In order to measure the product quality, we distinguish product and process product quality aspects. We defined metrics for the following quality aspects: *development progress*, *development performance, external quality*, *code quality*, and *build status*. These quality aspects can be combined to compute the indicators that will be used by the DevOps team to assess the quality of the monitored products, for example code quality can be used to assess product quality and development effectiveness.
At this stage of the project, the metrics that have been developed are mainly focusing on the project management and development activities. We plan to include quality of service aspects in the future.

b.3.1 Development progress metrics

These metrics are used by project managers to monitor the quantity of development tasks finished, the data comes from the GitLab tool.

- *Issue-closing Ratio*: Percentage of closed issues
  - Formula: `closed_issues / total_issues`, where
    - `closed_issues`: total number of issues with the state equals to "closed"
    - `total_issues`: total number of issues
- *Critical Issue-closing Ratio*: Percentage of closed issues that are prioritised as "critical"
  - Formula: `critical_closed_issues / total_issues`, where

        - `critical_closed_issues`: total number of issues with the state equals to "closed" and priority equals to "critical" (using issues' labels for both characteristics)
        - `total_issues`: total number of issues
- *OnSchedule-closing Ratio per phase*: Percentage of issues closed with a maximum of +-10% of deviation. This metric has been defined separately for each of the three phases defined in the Experis development process (Definition, Development, and Testing).
    - Formula: `onSchedule_issues/ closed_definitionPhase_issues`, where
        - `closed_definitionPhase_issues`=total number of issues with the state equals to "closed" and phase="Definition"
        - `onSchedule_issues`=total number of issues with the state equals to "closed" and `ABS(spent_time - estimate_time) <= (estimate_time*0.1).`
        - The 10% can be customised.

b.3.2 Development performance metrics

These metrics are used by project managers to monitor the development's velocity with respect to the estimations, the data comes from the GitLab tool.

- *OnSchedule issues*: Percentage of issues closed with a maximum of +-10% of deviation.
    - Formula: `onSchedule_issues/ closed_issues`
        - `closed_issues`=total number of issues with the state equals to "closed"
        - `onSchedule_issues`=total number of issues with the state equals to "closed" and `ABS(spent_time - estimate_time) <= estimate_time*threshold/100`.
        - `threshold`=10%
- Small Deviation issues: Percentage of issues closed with a schedule deviation between +-10 and +-20%
    - Formula: `smallDeviated_issues / closed_issues`, where
        - `closed_issues`=total number of issues with the state equals to "closed"
        - `smallDeviated_issues`=total number of issues with the state equals to "closed" and `ABS(spent_time - estimate_time) > estimate_time*threshold_lower/100) && ABS(spent_time - estimate_time) <= estimate_time*threshold_upper/100).`
        - `threshold_lower`=10
        - `threshold_upper`=20
- *Significant Deviation issues*: Percentage of issues closed with a schedule deviation higher than +-20%
    - Formula: `significantDeviated_issues / closed_issues`, where
        - `closed_issues`=total number of issues with the state equals to "closed"
        - `significantDeviated_issues`=total number of issues with the state equals to "closed" and `ABS(spent_time - estimate_time) > estimate_time*threshold/100).`
        - `threshold`=20

b.3.3 External Quality metrics

These metrics are used by the development team to control the quality of the software product from the point of view of reported defects/bugs, the data comes from the GitLab tool.

- *Critical Issue-closing Ratio*: Percentage of closed issues that are prioritised as "critical" (see Section b.3.1)
- *Bug-closing Ratio*: Percentage of closed bugs in the product backlog
  - Formula: `closed_bugs / total_bugs`, where
    - `total_bugs`=total number of issues with type equals to "Bug" (using issue's labels)
    - `closed_bugs`=total number of issues with type equals to "Bug" and state equals to "closed".
- *Critical Bug-closing Ratio*: Percentage of critical bugs closed with respect to the total number of bugs
  - Formula: `critical_bugs / total_bugs`, where
    - `total_bugs`=total number of issues with type equals to "Bug" (using issue's labels)
    - `critical_bugs`=total number of issues with type equals to "Bug" and and priority equals to "critical" (using issues' labels for both characteristics)

b.3.4 Code quality metrics

These metrics are used by the development team to control the quality of the written code, the data comes from the SonarQube tool.

- *Comment Ratio*: Percentage of files lying within a defined range of comment density.
  - Formula: `comments_good / comments_total`, where
    - `comments_good`: number of files considered well commented according to the comment's percentage.
    - `threshold_lower`=10%
    - `threshold_upper`=30%
    - `comments_total`: total number of analysed files.
- *Duplication Density*: Percentage of files lying within a defined range of duplication density.
  - Formula: `duplication_withinThreshold / duplication_total`, where
    - `duplication_withinThreshold`: total number of files with a duplication code below a concrete percentage.
    - `duplication_threshold`=10%
    - `duplication_total`: total number of analysed files.

b.3.5 Build status metrics

These metrics are used by the operations team to monitor the quality of the deployment process, the data comes from the Jenkins tool.

- *Build stability*: Percentage of successful builds in a certain period.
  - Formula: `success / total`, where
    - `success`: total number of builds with result equals to "SUCCESS" in a defined period
    - `total`: total number of builds in a defined period
    - defined period: last 300 days

c. Quality Use Case: Runtime Performance

c.1 Background

The runtime performance quality use case is centered around the performance of the embedded software of professional production printers. However, its principles are also applicable in other contexts, where embedded software is developed for complex machinery. For example, the KPI "Mean Prints Between Errors" mentioned below can be easily translated to other domains by replacing "Prints" by the items being produced, or the time between errors.

Functional printer quality KPIs are related to items like the quality of the prints produced (color gamut, resolution, reproduction of details, etc.), productivity (prints or square meters per hour), total cost of ownership (related to ink usage, service cost, media cost).

c.2 Data

The data is collected from two different sources. First, the machines in the field at customer sites report a large amount of structured system data, not only related to sensor and usage data, but also to the state of the software, including warnings and errors. Execution time of several components is also included in this logging. Second, the development data is gathered from the management environment (TFS or Azure DevOps Server) and the regression test results from the regression test framework.

c.3 Metrics

For the purpose of this document, we restrict ourselves to non-functional quality KPIs.

For general (non-functional) product quality, one uses KPIs like:

- MPBE (Mean Prints Between Errors).

  This denotes how much is being printed (in A4 prints or square meters) between the occurrences of an MRE (Machine Recoverable Error, solved by the machine itself, usually by restarting) or ORE (Operator Recoverable Error). This KPI can be compared with the common notion of MTBF (Mean Time Between Failures). These errors can have any cause (and usually the cause is from outside the software domain).

  There are also PE (Permanent Errors, to be solved by service), but these are not considered here. Here suitable KPIs would be the amount of service calls per year, the diagnosis and repair time, or the duration of service visits.

- Amount of software errors per square meter printed.

  While the previous metric referred to all kinds of errors, this one concerns only software errors. These are the errors logged by the software (and caused and observed by software). They usually lead to an MRE (Machine Recoverable Error). Target will be very near to 0.

The above KPIs are used during development and during deployment at customers.
During development we also have KPIs like:

- Amount of open and solved P1/P2/P3/P4 problems.

These are problems of any kind (with decreasing priority) reported during system tests. The priority is determined by the severity and chance of occurrence, as indicated in Table III below. For example a problem with ‘S2’ severity and ‘Regular’ chance of occurrence is given ‘P2’ priority. They are assigned to the discipline or system module that is causing them.

TABLE III
CHARACTERISTICS OF PROBLEMS (SEVERITY, CHANCE OF OCCURRENCE, PRIORITY)

Severity	
S1	Product inoperable, very important feature not working, crash
S2	Feature from specification not working and no work around available
S3	Feature from specification not working but work around possible
S4	Inconvenience or annoyance

Chance of occurrence				
		Frequency		
		Daily	Weekly	Monthly
Usage	Exception	Regular	Rarely	Rarely
	Common	Often	Often	Regular

Priority			
	Often	Regular	Rarely
S1	P1	P1	P2
S2	P1	P2	P3
S3	P3	P3	P4
S4	P4	P4	accept

Target for unsolved not acceptable problems at release would be 0.

Specific for software we use:

- Number of automatic tests.

  An important part of testing is performed automatically. Automated test cases are executed nightly at every build using a simulated engine. Ultimate goal is to test everything automatically and decrease the number of manual test cases to zero. Reporting is done using a dashboard and emailing on failed tests.

  Target for the amount of tests is such that they cover all functionality.
  Target for automation is 100%.
  Target for success is 100%.

- Software quality items resolved (here there are no real targets, but the items are reported).

  This addresses software design steps and removal of technical debt to keep the platform fit for the future. Each software sprint sets in its goal which quality items should be tackled.

For the runtime performance of the real time embedded software, we use:

- Overall CPU load (usage) during sustained operation.

  This is not the CPU load at any given point in time (this might easily be 100%), it is the CPU load measured during short intervals (e.g. of one millisecond) during which the printer is continuously working (printing) at full speed.
  It is comparable to what one sees in the windows task manager.

  Target is to keep this below 50-60% for the real time embedded software to guarantee the real-time deadlines.

- Specific for image processing:
  - Average and peak elapsed time for each image processing step.
    Should be below the available time.
  - Trend in these times during regression tests.
    Should not increase, unless explainable and it does not break the previous KPI.

  There are usually some 10-20 image processing steps involved (from receiving the image bitmap to placing the nozzle fire data in the buffer to send them to the print head).
  Some of these steps might be fused together in the implementation to obtain better performance.

  All deadlines should be met, thus the sum of the times of the steps in the pipeline should always be below the available time for each swath to be printed.

  The time needed might be affected by algorithms whose performance depends upon the image to be printed and the machine status (like nozzle failure compensation, which is dependent upon the number of currently failing nozzles), so this has to be verified by multiple tests.

d. Software as a Service (SaaS) Use Case

d.1 Background

Discussions with LaaS (Leadership as a Service) development team including the product owners, sales and other stakeholders, lead to the conclusion that the most valuable asset for planning the entrance to international Software as a Service (SaaS) market would be to have support for decision making which feature set should be implemented next. In this discussion, the difficulty is to balance the needs of the existing and potential customers and the development team's own ideas and wishes for further development.

d.2 Data

Data related to roadmap planning often resides in project management tools such as Trello, JIRA or Redmine. Product owners prioritize features, bugs, etc. in these tools to demonstrate the intended development order. Sometimes, the order might be changed by the developers because

of dependencies causing one less important (from the business point of view) item to be implemented before another higher value item.
Depending on the tool and agreed software development method, the developers estimate the items in these same tools. Agile software development methodologies advocate to use relative estimates and use complexity as a key indicator for the estimate. However, in real life, business often requires time estimates as business needs to know when something is ready for the delivery to customers. Sometimes other stakeholders crave for this information, too, for example to make promotion campaigns timely. Developers see time estimates often dangerous as they tend to be used to measure the performance of the teams or even worse, individuals. To make plans and lay out roadmaps for product development, a fast and realistic way to convert complexity estimates to time estimates is required.

For the needs of the SaaS use case it is most crucial to get the data regarding features and milestones from the project management system. This data can be augmented with other data described in Sections 2a -2c Software Quality use case to produce additional value. For example, data about technical debt could be used to augment the data nuggets, to demonstrate where refactoring efforts should be focused in order to provide the most value, if the schedule allows refactoring. Similarly, run-time performance data could be used to augment the feature development data to see the areas where improvement is needed. While these possibilities are nice to have, the focus of the visualizations will primarily be on facilitating the discussions between business and development and to decide the realistic roadmap.

d.3 Metrics

Next we describe metrics and their data source in the LaaS use case.

- Complexity of features to be developed.
    - Developers evaluate these in JIRA and add the estimate to the JIRA issue. Developers use various methods, e.g. planning poker to come up with a best possible estimate. Complexity estimates are relative to each other, i.e. tasks with the same rating are about the same in complexity. Metrics could be used to estimate the complexity of the planned change. However, Sjöberg et al. [6] shows that usually the only metric that correlates with the time spent on a change request is the lines of code. There is also evidence that the size of the code also affects the cyclomatic complexity [5]. In addition, these metrics can be applied only to changes to existing features, but not to a greenfield feature that does not exist at all. Thus, we use expert heuristics for this value.

- Development time (estimate)
    - Complexity estimates can be converted to time estimates as a batch by assigning a time frame for a whole that is formed of the tasks. We gather a set of tasks and create a release or milestone out of these and assign time value for the whole milestone. System then assigns time estimates to tasks automatically based on their complexity values. Now developers can make a sanity check if a task is impossible to do in a calculated time window. This time for a single task is shown in days. However, these time estimates should not be used for anything else than a feasibility check for the whole milestone.

- Development time (realized)
    - This can be gathered from the project management tools such as JIRA if time spent on the tasks are recorded there. This could help to make estimates more accurate in the future. Value can be in days.
- Costs
    - Essentially in SaaS development, the costs generated from the wages of the employees. Thus, time spent on a feature results in the cost of the feature. From the time estimates and realized development times we can estimate the costs and see the realized costs. Often the costs are key figures from the business point of view when making go/nogo decisions about the features.

- Value for the customer
    - There is no direct metric for what is valuable for the customer or for the development of customer partnership. We could ask customers to value the features according to their wishes, however, they rarely would like to use their time to do that. Thus, value for the customer is a rating that is given by a customer representative or account manager who looks after the customer inputs. If there is no customer representative or account manager available, the product owner him/herself can give an estimate on the value for the customer.
    In anycase, the unit of value is very abstract and represents more of a feeling than a solid data point. Sometimes, the customer or account manager could indicate the value in euros, but not in all cases. In most of the existing project management tools, this is not visible.
- Value of the customer
    - Not all customers are of the same value for the company developing a product. Some customers are more valuable than others. Some might bring in big turnover and sometimes it might be beneficial to get a customer for reference purposes even though the generated turnover is negligible. Thus, customers should be weighed when discussing the roadmap and which features for which customers we are going to develop next. Often in agile software development, it is the product owner's job to balance between the customers and to decide whose wishes will be prioritized to be the most valuable. This value is input by the Product Owner of the system as he or she should be capable of evaluating which customers are the most valuable for the company. In most of the existing project management tools, this is not visible.
- Overall value of the feature for the product
    - When we use 'value of the customer' to weigh 'value for the customer' values we can calculate the overall value of the feature (or a task) for the product being developed. This value can be used to discuss which features should be implemented next and to plan out the roadmap. In most of the existing project management tools, this value is not available. In the roadmapper tool we use the following formula to calculate the overall value of the feature (or task) for the product:

    ```
    Value of a task = for all customers SUM[(Value of the customer /
    SUM(Value of all customers) * Value for the customer] / Complexity
    of a task
    ```

- Value for developers
    - In addition to previously mentioned metrics for the product value, we can take into account the value task produces to developers. It might be that a task might decrease complexity, remove technical debt and consequently make the development faster (which gives business value faster). Often developers are the only ones who can see the value of such work. Thus, tasks should be estimated from the developer value point of view.

d.4 Additional data

Section d.3 described metrics that are useful for visualizing the product roadmap and to facilitate the planning of the roadmap. In addition, the visualizations could be augmented with all kinds of data to allow even deeper discussions about what should be worked on next. Here is a list of data that can be used to augment the aforementioned metrics to create new visualizations. However, these are not mandatory to enable discussions on the roadmap.

- Modules with most technical debt
- Modules with most code smells
- Runtime performance analysis
- Usage data from the system
- Modules with most bugs

e. Teaching Use Case

e.1 Background

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, teachers unanimously named one need above all else: the need to see how students are progressing during the course. Teachers needed tools to quickly and easily see which students are in the danger of falling behind and even out of the course, what tasks are bottlenecks for progress, and where students would need more support to be able to better accomplish the course requirements. While the need to see how students are progressing was unanimous, several teachers also hoped to find out the effort students used for their tasks and being able to see what topics were difficult to grasp. Finally, some participants hoped for visualizations of code metrics, of work distribution (among student groups), and on how the work pace differs between student groups.

e.2 Data

Real live data mainly comes from two data sources: GitLab code repositories and a MOOC-styled learning management system, Plussa. Teachers publish materials and 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. Additionally, in the future we will also integrate data from information systems holding grades from previous years, include data from SonarQube which is used on some courses, and look into utilizing data from Moodle.

e.3 Metrics

e.3.1. Progress

Based on most common use cases that arose from the interviews, we are basing our visualizations, and by extension the metrics, on the following assumptions of the workflow: 1) students need to complete weekly exercises, 2) each exercise has some maximum points defined that can be awarded to a student, 3) students submit an answer to an exercise by committing code. Further, we acknowledge there are differences in how points are awarded: 1) the maximum points that can be gained from an exercise varies, and 2) the minimum points required to pass an exercise varies. Finally – the number of exercises varies from week to week, and so does the distribution of points between exercises. For one week there may be several exercises awarding only small number of points each, and one exercise giving a large number of points.

For visualizations, the core metric to show student progress is the *current status* of a student. The current status of a student is shown in comparison to statuses of other students of the course (for easy comparison of how students statuses differ). Additionally, we will calculate the *average status*, i.e., the status of the "average" student on the course. If a teacher wants to focus on a small set of students, comparing to the average student will help keeping perspective. Finally, we will calculate the *expected status* of a student. The expected status is based on calculations from history data. We will define metrics for these in the following.

Current status
The current status of a student is multi-dimensional. The main dimensions (D) are **points**, **commits**, and **exercises**.
Cumulative points: fetch the points a student has gathered for each specific week and calculate the cumulative sum of points a student has gathered up to a specific week.

cumulativePoints$_n$ = $\sum_{i=1}^{n} gatheredWeeklyPoints_i$, where *n* = selected week

Missed points: calculate how many points the students have missed, i.e., what is the difference between maximum possible points and the points acquired by the student.

missedPoints$_n$ = $\sum_{i=1}^{n} (maximumWeeklyPoints_i - gatheredWeeklyPoints_i)$, where *n* = selected week

Relational points: calculate the percentage of points a student has gathered from all possible points available up to week *n*.
relationalPoints$_n$ =$\frac{cumulativePoints}{\sum_{i=1}^{n} maximumWeeklyPoints_i}$, where *n* = selected week

D2: Exercises

Cumulative exercises: fetch the number exercises a student has completed for each specific week and calculate the cumulative number of exercises a student has completed up to a specific week.

$$\text{cumulativeExercises} = \sum_{i=1}^{n} \quad weeklyExercises_i \text{ , where } n = \text{selected week}$$

Relational exercises: calculate the percentage of exercises a student has completed from all possible exercises given up to week *n*.

$$\text{relationalExercises}_n = \frac{cumulativeExercises}{\sum_{i=1}^{n} \quad maximumWeeklyExercises_i} \text{ , where } n = \text{selected week}$$

Relational exercises are closely tied to relational points. If a student has gathered a relatively small number of points but completed a relatively high number of exercises, it would indicate the student is often completing a large number of exercises that only offer a small number of points. As each exercise typically addresses a specific topic or learning point on a course, in this case the student will likely gather a superficial knowledge on a wide spectrum of topics. However, the student is likely not getting a deeper knowledge or willing to make distinct effort to complete tasks that would afford more points, as points typically reflect the estimated effort required to complete an exercise.

D3: Commits

Commits per exercise: calculate the number of commits a student has made for each exercise.

Average status

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. In effect, calculate average values for each dimension as given above: 1) the average number of points per student collected each week (D1), 2) the average number of points cumulatively collected per student up to a certain week (D1), 3) the average number of exercises completed per student (D2), and 4) the average number of commits made per task (D3). The "average" means average of all students participating in the same course implementation. In addition to "pure" average, we also calculate a threshold value to help identify those falling behind.

Expected status

Based on history data from previous implementations, we need to calculate how a student should be progressing at any given time in relation to a certain outcome (grade). The reference values are calculated from the average of collected points among students that have received the same grade on the previous course implementation. This approach aims at providing an estimation on how a status evolves when it is about to result in a certain grade.

The reference value is calculated from historical data originating from the previous course implementation by first defining the grade $g \epsilon [0, 5]$ and the course week number $w \epsilon [1, c]$ where c is the number of course weeks. Let

$$coursePartic ipantCount = \sum_{g=0}^{5} N_g,$$

where N_g is the number of students that received grade g. For each student $s_{g,i}$ that has received grade g, the student's cumulative points up until course week number w is $p_{s_{g,i}}(w)$. With these definitions, we get a week-wise reference value for each grade as follows:

$$referenceValue_{g,w} = \frac{1}{N_g} \cdot \sum_{i=0}^{N_g} p_{s_{g,i}}(w).$$

The expected grade for a student in the current implementation is determined by which grade has the closest reference value on the week of inspection. To view how the student has progressed during each individual course week, the above method is used by replacing the cumulative points by the sum of points collected during that course week.

e.3.2 Effort

Teachers have a clear need to see how much effort students use on the tasks given to them for two main reasons: 1) knowing the real effort used will allow teachers to adjust workload on the course to suit the credits given, and 2) knowing how much effort the course will need at different stages will help communicating the course demands particularly to students, but to other stakeholders as well.

Calculating effort is not straightforward. Still, even estimations based on real data are valuable. For effort we define metrics for the following:

1) Time used for each exercise (sum of used time per commits made for said exercise)
2) Time used each week (sum of time used for exercises on a given week)
3) Average, median, minimum and maximum hours spent per exercise
4) Average, median, minimum and maximum hours spent per week
5) Relation between used effort and number of completed tasks
6) Relation between used effort and expected grade

e.3.3 Level of difficulty

Teachers need information on what topics are difficult for the students to grasp. This is closely tied to previous sections on progress and effort: if a task requires a lot of effort, and many students fail to complete a task, then a teacher might look into the topic of the task to see why it is challenging for students. However, while metrics on progress are focused on the student, metrics on difficulty are focused on the exercises.

Metrics that indicate the success/failure rate of a specific exercise or topic should allow the teacher to 1) be able to spot bottlenecks and pain points in a course, and 2) use this information to provide more support and material on difficult topics.
Metrics are defined for the following:

1. Calculating the top percentage of tasks by 1) least effort used, and 2) highest level of completion
2. Calculating the bottom percentage of tasks by 1) most effort used, and 2) lowest level of completion

3. Calculating the average and median effort for 1) all exercises on the course, 2) all exercises for a specific week, and 3) all exercises on a given period
4. Calculating the percentage of students completing 1) each exercise on the course, 2) exercises on a specific week, 3) exercises on a given period.

e.3.4 Code metrics

In addition to completing or failing an exercise, teachers are eager to see the level of quality of submissions. For this purpose code metrics are desired. The actual code metrics will vary based on the planned learning outcomes of each course, and should be selected by each teacher. Possible metrics are:

- Percentage of code that is rewritten between commits
- Metrics gained from inspecting code, such as given by SonarQube. At Tampere University, Computing Sciences unit has access to SonarQube, and many courses are accustomed to using metrics from SonarQube.
- Metrics related to style guidelines.

To illustrate some possible metrics for a given course, we inspect the case of Programming 2. Programming two introduces the basics of objects and object-oriented programming, but does not really go into, e.g., inheritance. On this course, metrics related to style guidelines would include: complexity (depth of control structures), lines of code, length of functions (on average and maximum), number of functions, and number of public variables.

In addition to calculating and comparing metrics for and between individual students, it would be beneficial to calculate metrics and compare to the model solution.

e.3.5 Work distribution

Software development courses often have group assignments. When including group assignments, it is expected that all group members contribute to the assignment equally, but in reality this is rarely the case. Visualizations can help see who are not contributing, and how are students who contribute less progressing otherwise on the course.
Metrics:

1) Percentage of code commits made by each member of the group.
2) Issues opened/answered by each member of the group
3) New lines of code per commit by member of group
4) Contribution to README etc. documentation per person

e.3.6 Pace of work

There are two angles to visualizing the pace of work: 1) seeing the different points of time that particularly groups actually start producing code on a larger project assignment, and 2) how long does it take for a student to finish a certain assignment (different from effort). In both cases teachers are interested in seeing how the different paces of work affect the outcome on the course (i.e., how well assignments are finished with regard to given deadlines, and are there relations to points). While showing the different points of time when groups or individuals begin their work is left for visualization implementations, we may use metrics to calculate the timespan used for assignments, e.g., amount of time between starting an assignment and submitting it.

3. Conclusions

This deliverable has provided the metrics that are being used in the 3 different sub-cases of the Quality Use Case, as well as the SaaS and Teaching Use Cases. While it is possible that some of these metrics may be refined during the course of the VISDOM project, we consider them as a stable basis for further development of the visualization dashboard in WP3.

As a next step, we are working towards the combined usage of the different sets of metrics into the VISDOM toolchain and subsequently their visualization in the envisioned dashboards. While the five sets of metrics that were presented in this deliverable were developed independently of each other, there are opportunities of integrating them across the corresponding visualizations. This can be done in two ways. First, the metrics 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. For example the metrics on Technical Debt and Product Quality that are automatically extracted from source code and issue trackers can be easily fed into the dashboard of the Teaching use case.

Second, the metrics within the three sub-cases of the quality use case can be cross-pollinated to provide a wider perspective. For example the metric 'Software quality items resolved' of the Runtime Performance sub-case can be integrated with the metric 'Percentage of repaid TD' of the Technical Debt sub-case to incorporate both perspectives of technical debt in issue trackers and design quality issues. Throughout this deliverable, a number of concrete ideas to integrate metrics across the use cases were discussed. 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 the metrics that can be used in the VISDOM visualizations. However, the envisioned dashboard is, in principle, independent of the specific metrics used, and can be configured with any set of metrics, as long as they serve the purpose to gauge the health of the system or process. In the upcoming period we will demonstrate how the metrics presented in this deliverable, as well as other metrics can be combined and visualized within the VISDOM dashboard.

4. 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. V. Basili, G. Caldiera and D. Rombach, “The Goal Question Metrics Approach”, Encyclopedia of Software Engineering, Wiley, 1994.
4. 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
5. G. Jay, J. Hale, R. Smith, D. Hale, N. Kraft and C. Ward, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship," *Journal of Software Engineering and Applications*, Vol. 2 No. 3, 2009, pp. 137-143. doi: 10.4236/jsea.2009.23020.
6. Sjøberg, Dag & Anda, Bente & Mockus, Audris. (2012). Questioning software maintenance metrics: A comparative case study. International Symposium on Empirical Software Engineering and Measurement. 107-110. 10.1145/2372251.2372269.