D2.2.1
Documentations of the data models

Version history

Version	Date	Author (Partner)	Notes
0.1	31.08.2021	Kari Systä (TAU)	Initial structure
0.2	06.08.2021	Kari Systä (TAU)	Small fixes
0.3	07.09.2021	Aleksi Kytölä (VINCIT)	Roadmapper tool section initial draft
0.4	07.09.2021	Lou Somers and Roelof Hamberg (CPP)	Text for CPP section
0.5	07.09.2021	Kari Systä (TAU)	Added a candidate section for RUG demo
0.6	14/09/2021	Paris Avgeriou and Yikun Li (RUG)	Drafted RUG demo section
0.7	22/09/2021	Duc Hong (TAU)	Text for teaching case demo
0.8	30.09.2021	Ville Heikkilä (TAU)	Text and diagrams for teaching case demo
0.9	04.10.2021	Kari Systä (TAU)	Unification, summary, response to comments.
0.10	11.10.2021	Ville Heikkilä (TAU)	Cleaning teaching case demo text
1.00	12.10.2021	Kari Systä (TAU)	Cleaned up, candidate for the deliverable

1 Introduction

This document follows the target set in the FPP

> "In task T2.2 we define the data models needed by the demonstrators and their visualization types but also in the analysis (see T2.3). The data models are documented (D2.2.1) and supported by PoC implementations (D2.2.2)."

Thus, this document is organized according to the demos so that each demo is covered in a separate chapter. The discussions cover the first generation of demos, that have been shown e.g. at the first and second demo.

After presenting the demo cases some findings are summarized and directions for future research are given.

2 Teaching demo

2.1 Short description of the demo

In the teaching use case, the identified main stakeholders are teachers, teaching assistants and students. As these stakeholders have differing tasks and responsibilities on the same course, they have individual information needs and thus, are presented with different views into the data.

Based on a stakeholder interview, the main interest for teachers was identified to be seeing how students' progress over a course. Seeing the progress helps teachers to evaluate if there are, for example, some difficult topics or bottleneck exercises that require extra attention. However, teaching assistants are more interested in seeing the current status of students during each course week and to be able to identify students that might need help. Students, on the other hand, are likely interested to see how they are progressing in relation to the course goals and how much work they are required to do to achieve their personal goals.

The teaching demo is designed to show and pilot the above needs and builds around a programming course, taught in Tampere University. The course consists of weekly exercises, project works and an exam, and all course activities grant students points towards the final course grade. Since there are usually from 100 to 300 students on the course at a time, all the course material and exercise assignments as well as their submission system with automatic grading are published in a learning management system called Plussa (https://wiki.aalto.fi/pages/viewpage.action?pageId=159755451). Students solve weekly exercises locally and push their solutions into their own code repositories, hosted in the University's GitLab instance. Therefore, Plussa works as the main source of automatically collected data for the teaching demo, and GitLab serves as another data source.

2.2 Used data sources and their data models

As the first implementation for the data backend for the teaching demo case, a data fetcher implementation that collected all the course data from Plussa and GitLab was used. The collected Plussa data included metadata about the exercises in the course and points data about the students participating in the course. In the course, the exercises were collected into modules which correspond to a calendar week with a total of 14 weeks in the course. The points data included exercise specific points that the students received as well as all the submissions the students had made for each exercise. The

collected GitLab data included the exercise specific commit data that each student had made.

The data fetcher preprocessed the collected data to conform to the data model described in figure 1. The course points part of the data model matches closely to the data model used in the Plussa REST API. Thus, most of the preprocessing involved tying the commit data to the exercises. This preprocessed data was then stored to Elastic Search database and used as the input for the visualizations. Only data from students who had given their permission for the data collection was collected and all the data that could identify the individual students was pseudonymised in a way that only someone who has teacher access to the course in the Plussa system can find out the actual student identities.

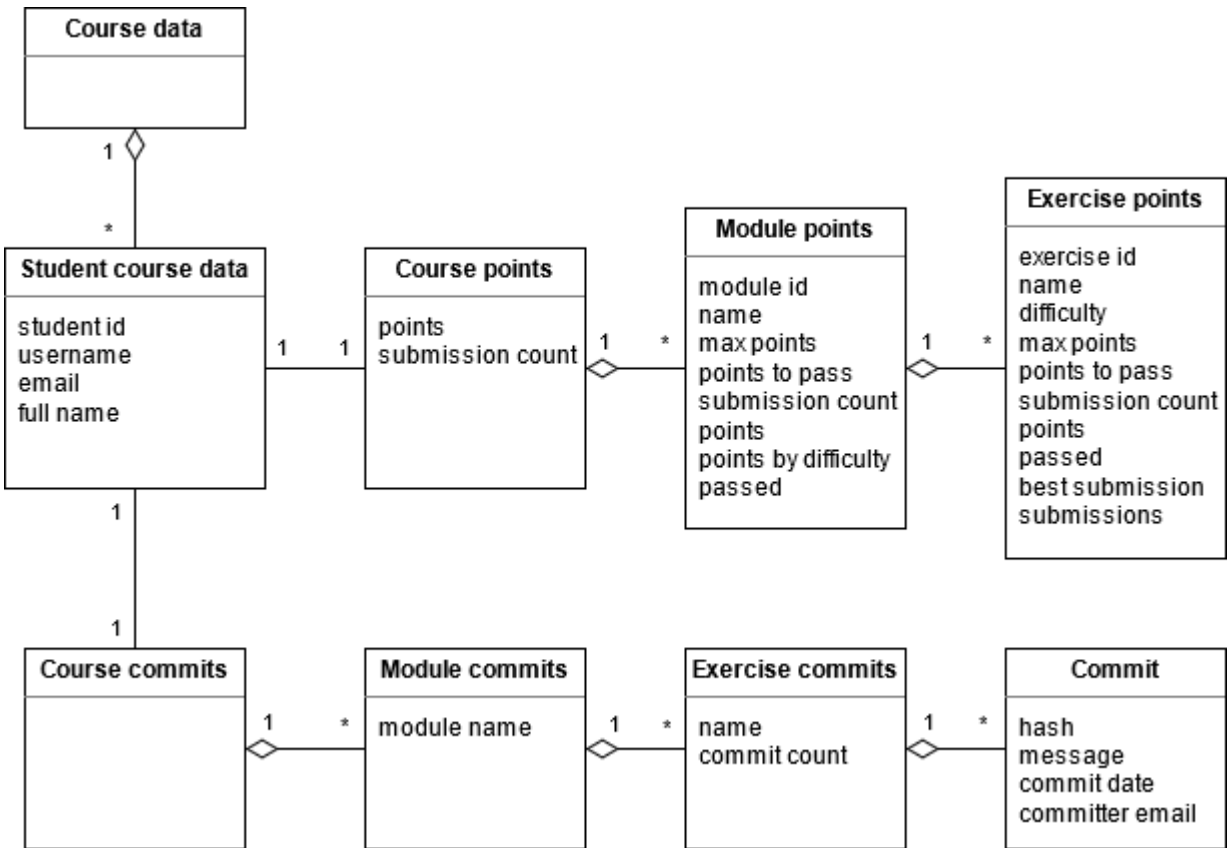

Figure 1. The data model representing the current situation for each student in a course in the teaching demo. Some additional metadata attributes that are not displayed here were also included in the data.

The data shown in Figure 1 is already rather far preprocessed for visualization; the intent to follow the progress of the students has clearly affected the design. The actual domain elements to be visualized are the structured representation of a student's tasks (module, exercise), evaluation of the exercises and commits.

To make comparisons and predictions on how the students were progressing on the ongoing instance of the course data about the earlier instances of the course is needed. A summary (aggregated) data from earlier implementation is thus given to the visualization. The data model of this is shown in Figure 2.

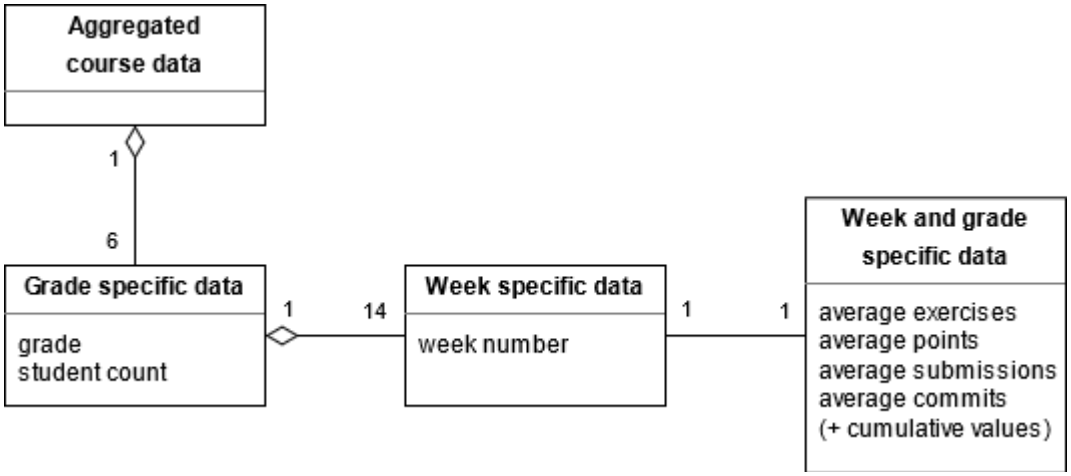

Figure 2. Data model used to represent the aggregated data from a course in the teaching demo. It contains weekly information about performances of students grouped by the grade (0, 1, 2, 3, 4, or 5) they received from the course.

This approach only works if the previous course instance has a similar module and exercise structure as the new instance. In the teaching demo case the course instances vary slightly but not enough to cause issues for this approach of comparing points and other values between course instances.

Together the data models represented in figures 1 and 2 are given as input to the visualizations. To be exact, for the course that is being visualized, one Course data document following data model from figure 1 and one Aggregated course data document following data model from figure 2 is the combined input data. The contained information includes the student specific data list outlined in the D2.3.1 section 3.e.3, though the cumulative and average values (items 3, 4, 6 and 7) are not given explicitly:

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

To be able to visualize the student data using the progress metrics defined in D2.3.2 section 2.e.3 the input data requires some further processing. For example, the module specific cumulative, missed and relational points are calculated for each student at this point. In the demo this processing is done in the visualization code each time visualization is started. The data model described in figure 3 represents the data model created and used by visualizations after they have combined the data from the ongoing course with

the aggregated data from the previous course instance and done the required data processing.

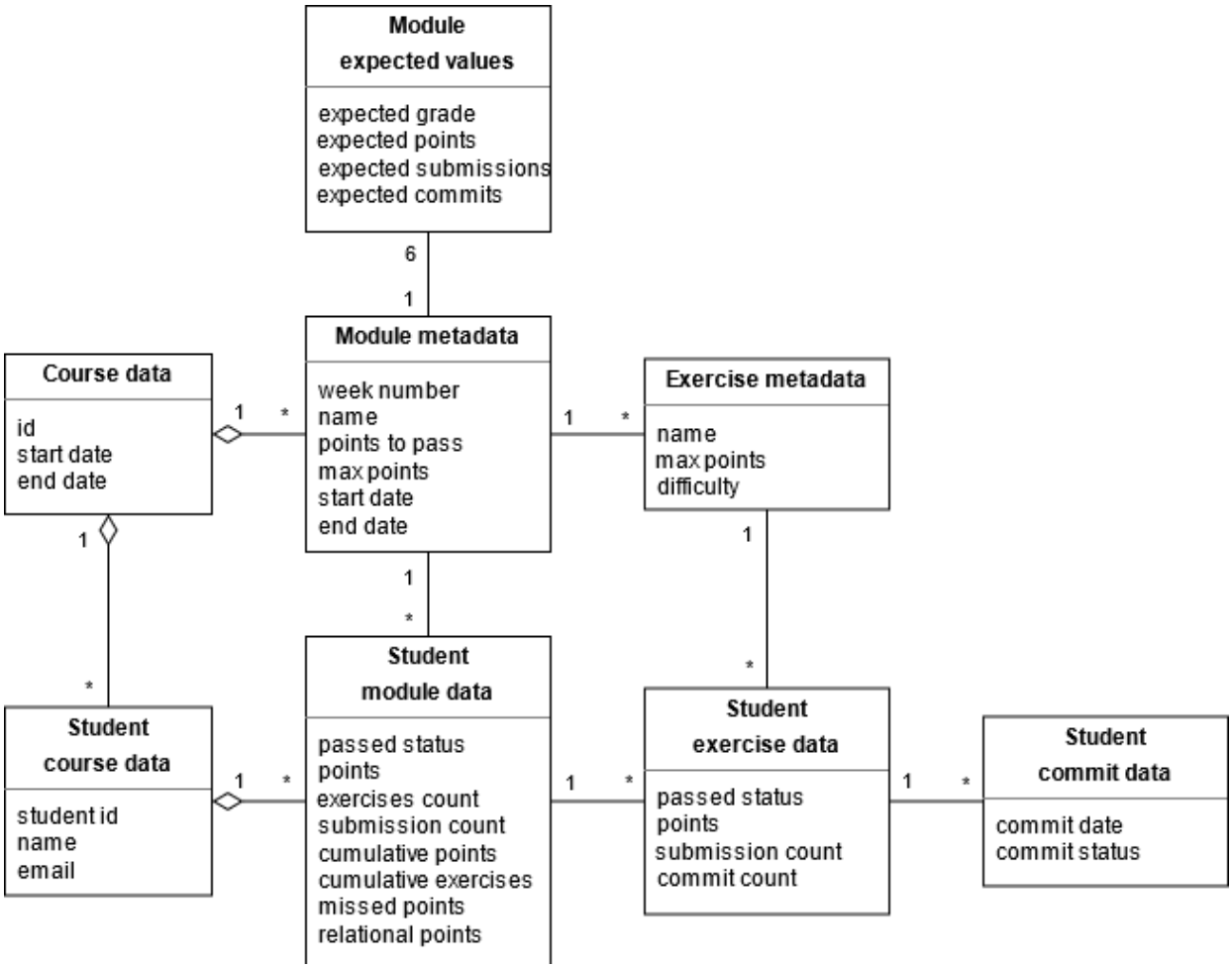

Figure 3. Data model for the fully processed data that visualization applications use when visualizing the course data in the teaching demo. Note, that the given data model might not match exactly to what is used in the visualizations but includes the same content and links between the different items.

2.3 Lessons learned and guidance for next phases

The first demo concentrated on quick development of visualizations, and the data backend was not paid too much attention. However, while implementing the visualization we discovered the following problems:

- In some visualization use cases it is natural to use time measures relative to the start of the project – sometimes it is more natural to use absolute calendar time.
- The data was complemented with metadata, and visualization code gets easily fixed to ad hoc design of that metadata. Metadata needs should be part of the design of data models.
- There is a "hyperparameter", which is the time length of each "round". This time length would be the unit of the x-axis in visualizations. With this parameter, we can shrink or extend the view of data, which can provide a better way to analyze the

data. This also brings the flexibility for visualizations, because the current solution has the fixed time unit, which would not work appropriately when the time unit changes

- Although the attribute-mapping method is the central factor to generate the desired visualizations, the system configurations can increase the degrees of freedom that the visualization would change the data representation significantly. With the combination of attribute-mapping configuration, and the system configuration, we can map a configuration to one of the previously developed visualizations.
- Not all attributes can be mapped to any visualization variables. The property of attributes can affect the way it can be displayed.
- The data adapter is important in the software, as without processing the raw data first from the data fetcher, the time for the client for processing the raw data every reload is noticeable.
- The creation of the combine data (shown in Figure 3) is a time-consuming operation and slows down the initialization of the visualization.

3 Roadmap tool

3.1 Short description of the demo

VISDOM Roadmapper is a tool prototype that visualizes the roadmap of a SaaS product to facilitate the discussion of what features or tasks should be completed next. The tool uses customer and task models alongside work and value ratings as data to generate visualizations to aid in the planning of the project roadmap.

3.2 Used data sources and their data models

The ER diagram below represents the data model that the Roadmapper tool uses internally. Tasks can be imported into roadmaps from supported project management tools (Trello, Jira) or they can be added manually.

The roadmap and customers are created by the Project Owner user. More users can then be invited with chosen roles and can optionally be assigned as representatives for customers.

Users submit value / work ratings for tasks via the applications web user interface.

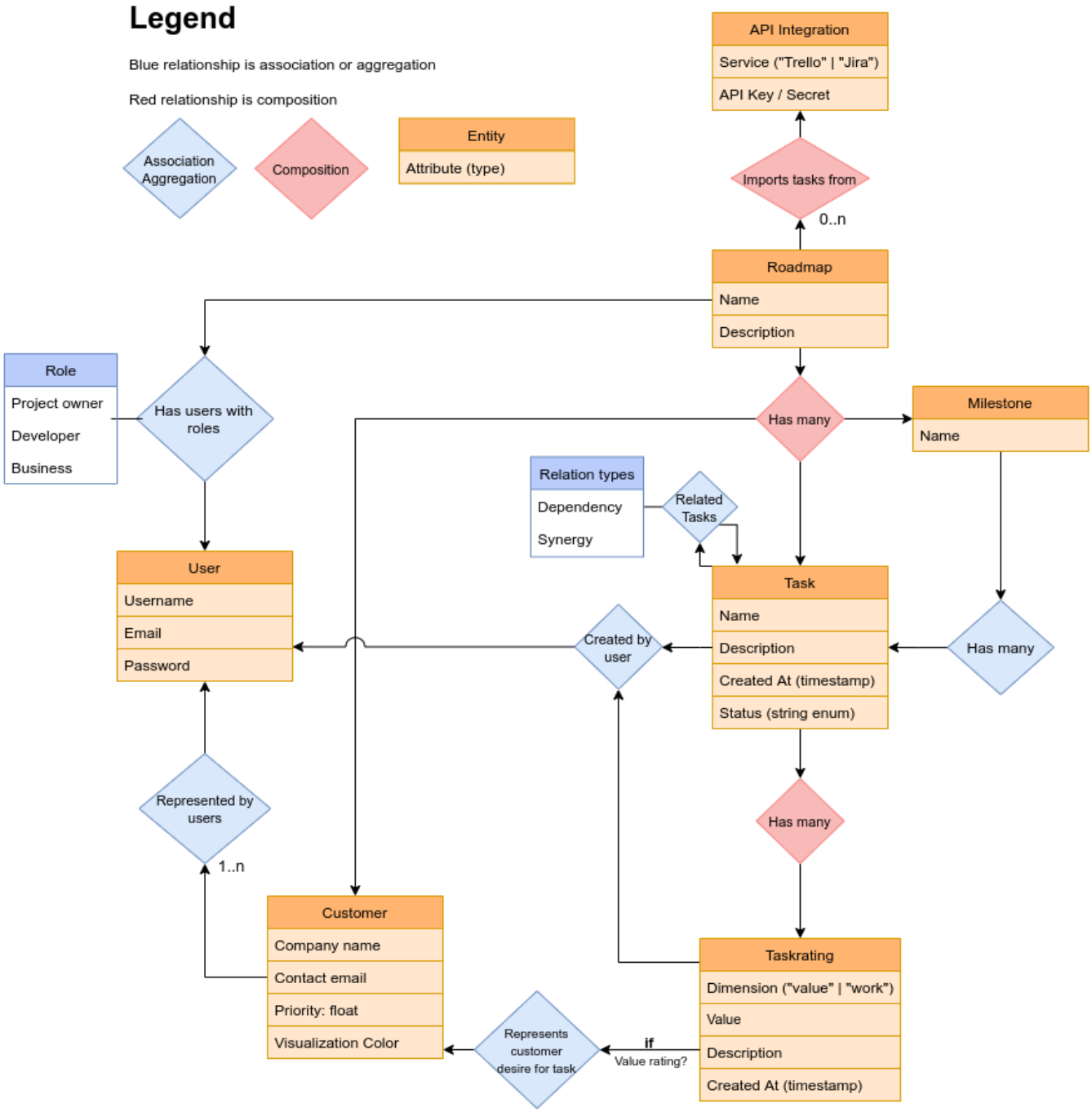

Figure 4. Roadmapper data model entity-relationship diagram

Some of the concepts in the above Figure relate to the implementation of the tool. The visualized domain data include the following four elements: Roadmap, Milestone, Task, Task Rating, and Customer.

3.3 Lessons learned and guidance for next phases

It's important to have a distinct roadmap from the project management tools available. Developers tend to fall in the mode of thinking that the roadmap visualization is another view on the Trello or JIRA tickets, but it is a tool for discussion on the business goal, the big picture and technical dependencies. To discuss the latter we need to visualize the dependencies somehow.

Often roadmapping is guided by the upcoming deals or the needs of the existing customers. However, as the development of a SaaS system proceeds, we need to start tackling the technical debt. Thus, it would be beneficial to add visualization of technical debt payback on the roadmap. This helps to see the overall amount of the debt and see how much it slows down the future development of new features. On the other hand, it calms down the developers as they see that something is being done to handle the technical debt.

4 Quality demo at Canon Production Printing

4.1 Short description of the demo

There are two dashboards involved:

1. Runtime Performance Trend Visualization
2. Visualization of example-driven development to track progress and regression over time

The first one, the visualization of runtime performance trends is described in more detail in D2.3.1. It shows for each software build the performance of the different image processing algorithms as measured in the automated regression tests, corrected for the speed of the test machine used.

The second one is a visualization dashboard to track progress and regression over time of software development. The specifications of the functionality to be developed is example-driven, largely inspired by behaviour driven development. The examples serve to illustrate the main functionality on the one hand, and provide an overview for this, while the examples are at the same time executed as tests on the developed software for each build. The specifications in so-called feature files evolve with the developed software product and are guaranteed to be in sync with each other.

4.2 Used data sources and their data models

The visualization of the runtime performance of image processing software uses execution time data from the functional logging that is written during the regression tests (as described in D2.3.1).

The data model can be visualized as in Figure 5:

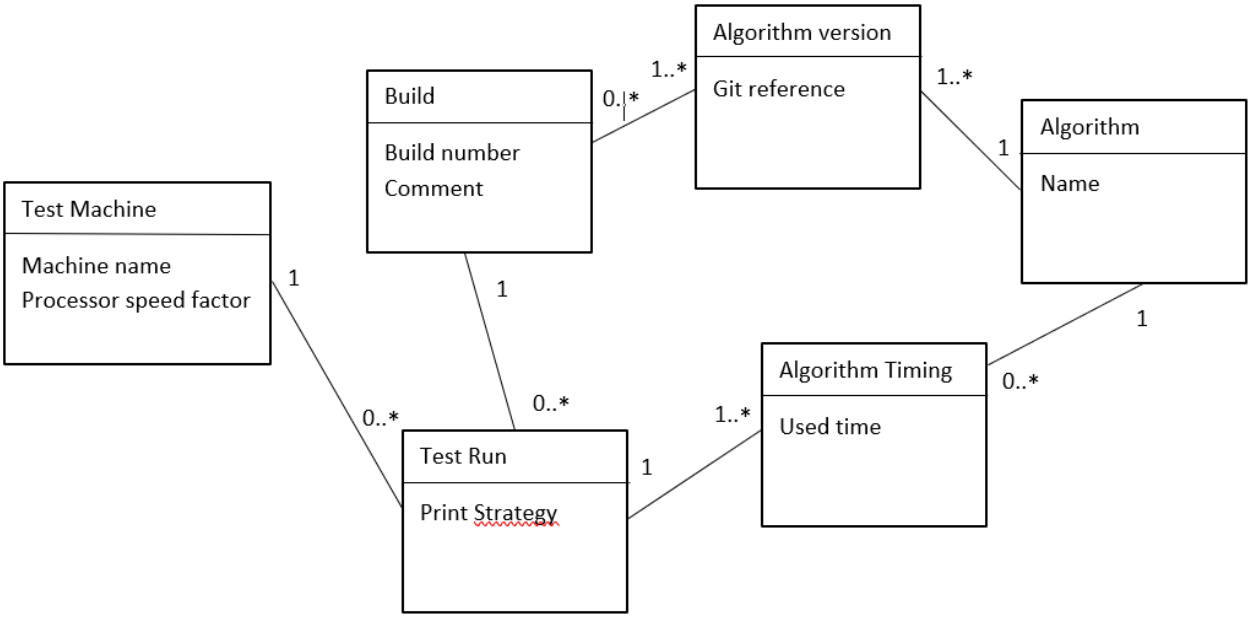

Figure 6: data model of the runtime monitoring in the Canon demo.

After each build, a number of automated regression tests will be run. For each regression test, the execution time taken by the different image processing algorithms is logged. Also the name of the machine on which a test runs is logged to be able to correct for the speed of its processor. The specific versions of the algorithms used in each test run can be traced back via the build number.

In other words, the data to be visualized include Algorithm, its Timing, and Version. In addition, the Build, Test Run and Test Machine are used, too.

The visualization of the example-driven development uses data of the development repositories as well as data of build-time test results. The data model can be sketched as in Figure 7.

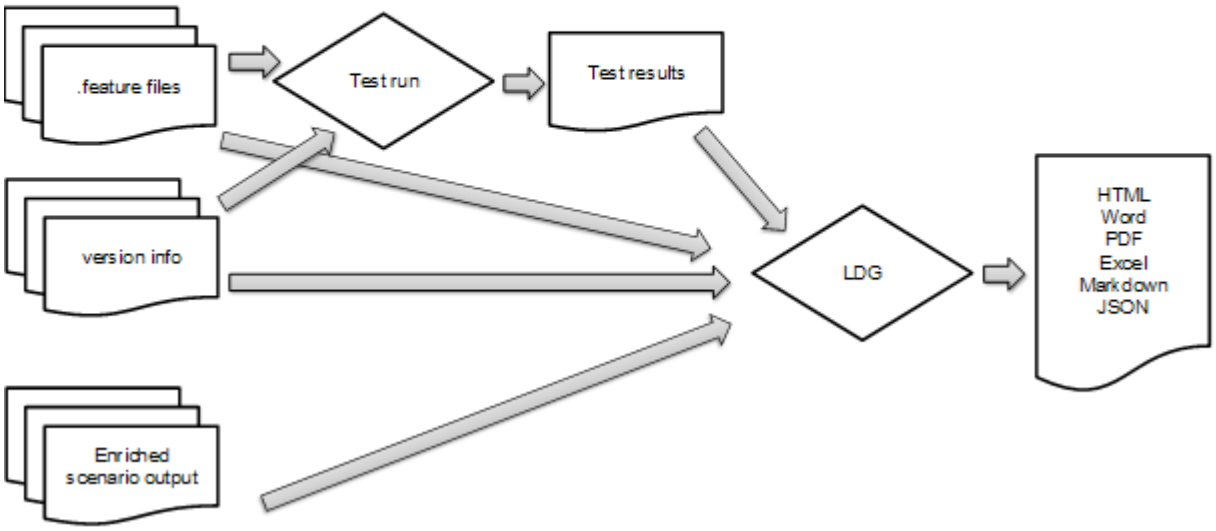

Figure 7. Use of development repositories in the Canon case.

The Living Documentation Generator (LDG) combines multiple inputs to a single dashboard overview, which is a collection of static HTML pages in the first version. The .feature files contain plain text specifications in a structured language format also suitable for test definition. The enriched scenario outputs are higher level explanations of the supported scenarios, expressed in free Markdown format. Generated test results after each build are combined by the LDG to update the dashboard.

4.3 Lessons learned and guidance for next phases

In order to increase the speed of tracing back the possible causes of sudden performance drops, it is important to consider the performance visualization graphs on a regular basis and keep track of changes in the algorithms.

The first version of the development dashboard is being used by the product owner. The first request for useability is to have a slightly more fine-grained status of the features in the overview to enable development tracking in a more natural way. Full traceability to work item definitions in TFS is foreseen for the future.

5 Quality demo by Experis

5.1 Short description of the demo

The quality demo builds around a DevOps team that develops a product named PHE developed in Experis (that consists in health & wellbeing services for their employees) composed of two projects. These two projects consist of: (a) an application (APP) to be

deployed in the final users devices (named phe); and, (b) a set of services to be deployed in a server that are consumed by the APP (named phe_server). Following DevOps strategy, the team is composed of two connected teams, development team that is responsible for the development of both products and operation and a team that is focused on the services deployment. The source code and the backlog are managed using GitLab. GitLab data is complemented by static (source) code analysis provided by SonarQube and the continuous integration tool Jenkins. [From D3.4.2]

5.2 Used data sources and their data models

The data sources used are data from the PHE and PHE Server projects, Experis projects dealing with health and wellness services for their employees.

The applications contain user data that is collected through activity wristbands, thus generating a high volume of data to be processed.

The organization of this data is based on a relational BB that organizes this data to later be processed and displayed.

Within the Visdom project, this data is analyzed through the solution proposed as demonstrated since the relevant information is extracted from a DevOps environment, which is treated and displayed through the QRapids tool where you can see the graphs of each of the projects addressing issues of interest such as Bugs ratio, well defined issues, comment ratio, duplication density, test performance, passed test percentage, etc.)

Below are the data sources and the quality model from which it starts.

Data sources:

Quality Aspect	**Factor**	**Assessed Metric**	**Description**	**Calculation**	**Raw Data**	**Data Source**
Maintainability	Code Quality	Complexity	Files below the threshold of cyclomatic complexity (10 by default)	`Density of non-complex files =Non-Complex filesTotal number of files` where a file is complex if the <cyclomatic complexity> per <function> is greater than <10>.	*Cyclomatic complexity* per *function* of each file, *total number of files*	SonarQube

Maintainability	Code Quality	Comments	Files whose comment density is outside the defined thresholds (by default 10%-30%)	Density of commented files =Commented filesTotal number of files where a file is commented if the < density of comment lines> is between 10% and 30% Density of comment lines =Comment lines(Lines of code + Comment lines)	*Density of comment lines* and *lines of code* per each file	SonarQube
Maintainability	Code Quality	Duplication	Files below the threshold of duplicated lines percentage	Absence of duplications=Files without duplicationsTotal number of files where a file has no duplications if the < density of duplication> is less than 5% Density of duplication $=Duplicated\ lines$Lines	*Duplicated lines* and *lines of code* per file	SonarQube
	Software Stability	Bug density	Ratio of open issues of the type bug with respect to the total number of issues within a customized timeframe	Ratio of bugs $=Number\ of\ open\ issues/in\ progress/re-opened\ of\ type\ "bug"Number\ of\ open\ issues/in\ progress/re-opened * 100$	Total number of *issues* (a.k.a. tasks) per *status* (e.g., open, done), *type* (e.g., bug, maintenance, feature), and *timeframe* (e.g., current/last month or current/last sprint)	Jira, GitLab, Redmine, Mantis
Productivity	Issues' Velocity	Resolved Issues' Throughput	Density of issues whose resolution didn't take longer than the defined duration threshold	$=Number\ of\ issues\ resolved\ under\ the\ duration\ thresholdTotal\ number\ of\ issues\ in\ a\ given\ timeframe$ where the duration threshold is user defined	*Total number of issues* with *resolved status* (e.g. sprint, week, month) and *total issues* in *timeframe*	Jira, Mantis, GitLab

Development Speed	Avg. Automation Duration	Density of automated tests that didn't take longer than the defined duration threshold	$=\frac{Number\ of\ automated\ tests\ under\ the\ duration\ threshold}{Total\ number\ of\ automated\ tests\ in\ a\ given\ timeframe}$ where the duration threshold is user defined	Every *automated tests* with *test duration* under a threshold, and *total number of automated tests* in *timeframe* (week, month, sprint, etc.)	Jenkins	Avg. Automation Duration
Development Speed	Build performance	Density of daily builds that didn't take longer than the defined duration threshold	$=\frac{Number\ of\ builds\ not\ taking\ more\ than\ the\ duration\ threshold}{Total\ number\ of\ builds\ in\ a\ given\ timeframe}$ where the duration threshold is user defined	Every *build* with *build duration* under a threshold, and *total number of builds* in *timeframe* (week, month, sprint, etc.)	Jenkins	Build performance

As for the extracted data model, it considers the possibilities based on the data sources as explained in D2.3.1 which details the set of indicators aggregated in a quality model where each level of the quality model provides an assessment of quality (these are Quality metrics, quality factors and strategic indicators).

This demo is centered around the quality model composed of the above metrics. The model as well as the visualization is based on the data of three categories. 1) static code quality consists of Complexity, Comment Density and Number of Duplicated Lines, 2) dynamic quality measured from the Reported Issues, 3) development speed as rate of Automated Tests and number of Daily Builds.

Quality model

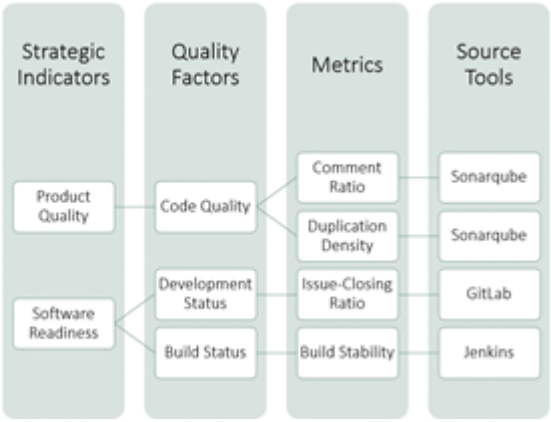

Figure 8. Quality models in the Experis demo.

As can be seen in the quality image of the model, it is based on data sources from GitLab, Sonarqube and Jenkins where GitLab data is complemented by static (source) code analysis provided by SonarQube and the continuous integration tool Jenkins (aspect that was highlighted in D3.4.2).

On the other hand, as already explained in general terms in D2.3.2 metrics are being used to measure the quality of the product, there are metrics for the quality aspects (which are development progress, performance, external quality, code quality and compilation status). These quality aspects can be combined to calculate the indicators used by the DevOps team to assess the quality of the monitored products.

5.3 Lessons learned and guidance for next phases.

It should be noted that it has been possible to contrast the results with the original data and that a more global vision of the results can be appreciated, i.e., where before the focus was on individual issues, now there is a global vision that provides information with context of the state of the developments, being able to know that, for example, if there is technical debt, what it may be due to or that if the times are delayed with respect to the planning, it can be easily observed where the main problems are to be found. In this way we have obtained tests aligned with the DevOps philosophy where the shared and horizontal information gives an extra in terms of development control as well as the anticipation of solutions before the problems become evident.

6 Technical Debt demo at RUG

6.1 Short description of the demo

The demo tool consists of two components: the SATD classification tool and the SATD visualization tool. The SATD classification utilizes a machine-learning (CNN-based) approach to automatically identify SATD from source code comments, commit messages, pull requests, and issue tracking systems. The SATD visualization tool is a web-based

application that uses the output from the SATD classification tool, processes it and visualizes the results.

The tool is meant to be used by members of the development team in order for them to obtain an accurate view of the system's internal quality. It can show both snapshots of the system's technical debt, as well as its evolution over time, allowing software engineers to look both at the big picture and zoom in on individual details (e.g. specific technical debt items). It can be used during discussions about the software internal quality, and particularly about prioritizing tasks to improve such quality, e.g. by means of refactoring. The tool uses multiple sources (issues, pull requests, commits, code comments) in order to provide a more complete picture of the system's technical debt: while none of the sources in itself is comprehensive in technical debt identification, by combining the sources we can obtain a composite and rich view of internal quality problems.

6.2 Used data sources and their data models

The data model is illustrated in the diagram below:

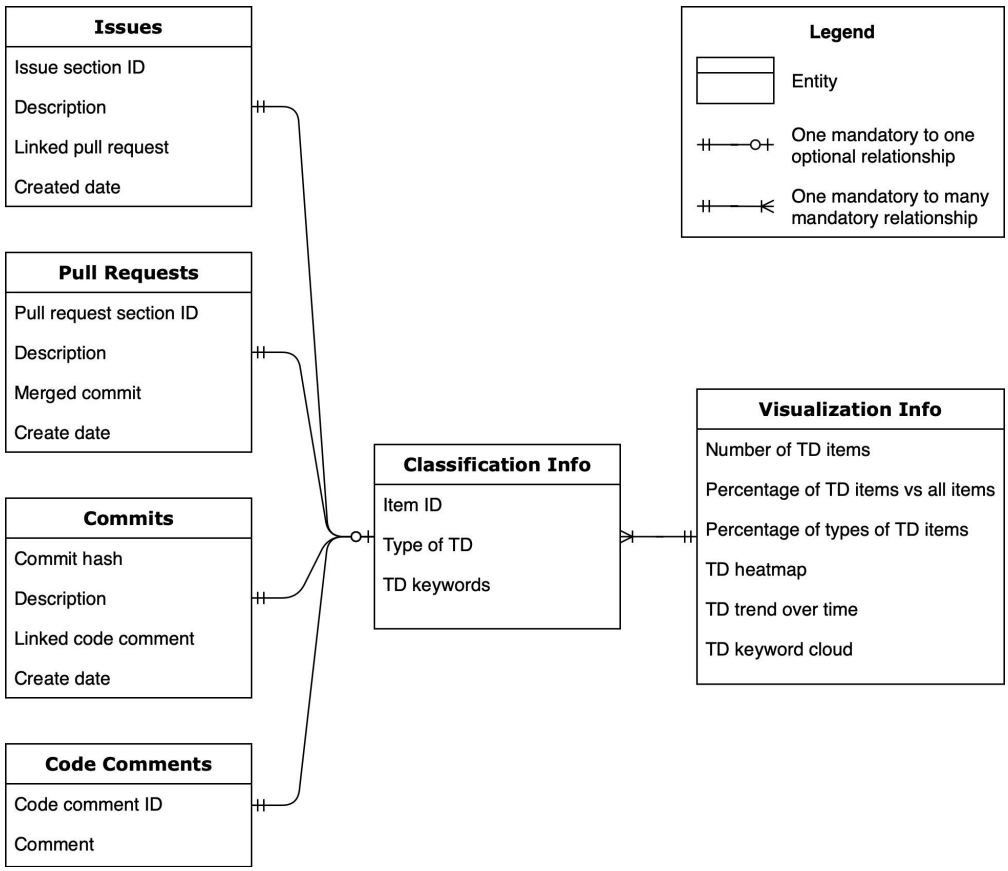

Figure 9. Data models of the Technical Debt visualization by RUG.

6.3 Lessons learned and guidance for next phases

Some preliminary usage of the tool by software practitioners has demonstrated that the tool achieves its goal relatively well. The different visualizations provided allow both for macroscopic and microscopic views on technical debt. Different stakeholders have been able to use it without facing issues and were able to provide us evidence on the merit of the tool. They also gave us some points for improvement. First, our SATD classification approach does not differentiate between unpaid SATD and paid SATD. It is important to extract all unpaid SATD items from different sources. Then developers can measure them and prioritize the SATD repayment. Second, our approach does not tackle measuring the identified SATD. Technical debt can be measured in terms of the risk it entails. Using this property of SATD, we can prioritize certain items over others, indicating to developers to focus more on what is important. We plan to improve on these two points in the next iteration of the tool, as well as working further on the usability to ensure it can be used off-the-shelf by software engineers.

7 Discussion and Conclusions

The five demos used the following domain concepts to be visualized:

1. Teaching: Module, Exercise, Exercise Grading, and Commit. (4)
2. Roadmap tool: Roadmap, Milestone, Task, Task Rating, Customer (5)
3. Canon: Algorithm, Timing, Build, Test Run and Test Machine. (6)
4. Experis: Complexity, Comment Density, Number of Duplicated Lines, Reported Issues, rate of Automated Tests and number of Daily Builds (6)
5. RUG: Issues, Pull Requests, Commits, Code Comments (4)

The total number of input data types is 25 out of which only few (Issue, Commit, Task) are shared among the demo cases.

The demos also vary between each other on how data has been processed for the visualization. In the teaching demo case, the data is already pre-processed for the visualization purposes. On the other hand, Experis and RUG demos assume that the data is analyzed and visualization shows the analyzed data. Actually, in all of our demos the visualizations want to use pre-processed data, but the nature of preprocessing varies.

These findings support the central idea of the VISDOM architecture: instead of aiming at a general unified data models, we should define purpose-specific data-adapters and maximize their reusability. As stated in D2.5.1 "***Data adapters*** *implement a two-way adaptation. Firstly, it provides visualizations a uniform data model regardless of the data source. As an example, the data adapter may facilitate visualization for similar "ticket" data regardless of it being fetched from Jira or Trello. Secondly, data adapters may provide different data for different types of visualizations."* In addition to that, the experiences from the first generation demos indicate that the adapters should be built around some analysis and pre-processing so that visualization gets data that more closely matches the actual concepts to be visualized.