

Requirement Management Process

Sibylle Peter <sibylle.peter@canoo.com>, Canoo Engineering AG, Franz-Josef Wiszniewsky <franz-josef.wiszniewsky@canoo.com>, Canoo Engineering AG, Alexander Nehmer <a.nehmer@science-computing.de>, science + computing ag

Version 1.3
Mai 19, 2015

Table of Contents

- 1. License 2
- 2. Abbreviations, Glossary and Roles 6
- 3. Participants 7
 - 3.1. Steering Committee 7
 - 3.2. Product Owner/Product Manager 7
 - 3.3. Developer team 8
 - 3.4. Architecture Committee & Quality Committee 8
 - 3.5. Test team 8
 - 3.6. Summary 9
- 4. Requirements 10
 - 4.1. Requirement template for Epic/Story 10
 - 4.2. Bug Reporting 12
- 5. The life of a requirement 13
 - 5.1. Life cycle in sponsored projects 13
 - 5.2. The classic open source life cycle 17
- 6. Requirement/Issue Management Tool (Jira) 18
 - 6.1. Workflows 18
 - 6.2. Security 19
 - 6.3. References 21
 - 6.4. Other Jira Projects 21
- 7. Direct comparison of two organisational approaches regarding requirement management 22
 - 7.1. Suggestion by the requirements management process service (Canoo) 22
 - 7.2. Suggestion by Sven Wittig 24

This document explains the proposal and includes the discussion of the requirement management process of the OpenMDM project by demonstrating the life cycle of a requirement request. First the participants in the process are named. Afterwards the different stages of a requirement towards implementation and release are outlined. The thoughts and process presented here are proposed by Canoo Engineering AG and include the discussions and conclusion of the requirements workshop on December 8, 2014 as well as follow up meetings on april 22 and mai 19 2015.

Chapter 1. License

Eclipse Public License - v 1.0

THE ACCOMPANYING DOCUMENT IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

- a. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
- b. in the case of each subsequent Contributor:
 - i. changes to the Program, and
 - ii. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

- a. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be

covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

- b. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.
- c. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

- a. it complies with the terms and conditions of this Agreement; and
- b. its license agreement:
 - i. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
 - ii. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
 - iii. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
 - iv. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

- a. it must be made available under this Agreement; and
- b. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that

reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR

DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Chapter 2. Abbreviations, Glossary and Roles

- AC: Architecture Committee
- EWG: Eclipse Working Group
- QC: Quality Committee
- PO: Product Owner/Product Manager: the person who makes the decisions about priority and guides the implementation.
- SC: Steering Committee
- RSO: Requirement Service Owner. It is planned to have a requirements management service. If it is also the case to have a centralized PO, then these two roles are identical.
- RO: Release Owner. A member of the Steering Committee or the sponsor of a contribution.
- Sponsor: The responsible person who is funding a work package. This person also acts as a work package PO, resp. can delegate this role to somebody.

Chapter 3. Participants

The following people may influence or are directly involved in the processing of any requirement and are therefore briefly described:

3.1. Steering Committee

The committee members can file requirements like everyone else but are not directly involved in the concrete realization of a desired functionality. Instead the committee sets boundaries by defining the overall goals for each release. The product increment as being the result of each release is approved or denied as a whole by the steering committee.

3.2. Product Owner/Product Manager

Product Owner is the (see [Scrum methodology](#)). This role is responsible for the definition and prioritisation of the functional and non-functional requirements of the product. She is responsible for shaping the vision together with the stakeholders and make sure that everybody building the product understands it (interface between stakeholders, sponsor and development team). The product owner has the competence to decide what should be build to create as much value as possible as quickly as possible. In Scrum requests for and issues of a product are managed in the so called **backlog** - the product owner is in charge of maintaining and prioritizing the backlog.

3.2.1. What does that mean for the OpenMDM EWG?

To get a successful product we think that is crucial to have a dedicated person to fulfill this product owner role. We propose to choose an independent person and pay it using a work group service. Very often the engagement of a product owner is underestimated, but since she is basically responsible for success or failure of the product, it is essential that this person has enough time and knowledge. The OpenMDM product owner is responsible to collect and evaluate requests from the community (backlog management). She is also responsible to bundle and select the most valuable and efficient issues needed to reach the roadmap goals defined by the steering committee. Ideally the product owner escorts the development of packages/blocks of development - as she is the only person who can make quick decisions (short feedback cycle). The OpenMDM product owner reports to the steering committee and signs off the delivered product increments. She is the primary person which defines **WHAT** (functional requirements) is implemented. But it is important that the product owner is a good team player since the architecture committee and the quality committee define the **HOW** (non-functional requirements) of the product - and the team (service provider) is in charge to estimate the effort and to develop the requirements in the required quality in the simplest way with the least effort possible.

Discussion of the workshop 8-12-2014 Sven Wittig: Sven only sees the need of such a role in the collecting and bundling of requests from the community (requirements service owner). The bundles/packages then are presented to the steering committee and provided they get a sponsor, are

then implemented according to the architectural and quality guidelines. The escorting of the service provider is the sole responsibility of the sponsor. Basically Sven Wittig assumes a **bottom up** (community driven) approach only.

Others (Ulrich Bleicher, Andreas Benzig, Sibylle Peter) While they see a possibility of a release owner which is responsible for the implementation of a work package, they still think a global OpenMDM product owner is needed to provide conformity and congruence for the product. They think it is necessary to have a **top down** (steering committee provides initial vision and roadmap for the product) as well as a **bottom up** approach at least in the beginning.

This discussion led to another interesting and important question, again from Sven Wittig: What is the product? Is it only a set of building blocks, loosely coupled, configurable and working together or is the product a system consisting of loosely coupled building blocks, which allows the community users to execute the most common shared use cases?

IMPORTANT

these questions (what is the product and which approach(es)) should be discussed in the steering committee asap.

3.3. Developer team

The developers are responsible for the concrete implementation of the specified requirements. The team supports the product owner by pointing out costs, dependencies and risks linked with the realization of the requests. The developers are responsible for creating the shippable product at the end of each release iteration. In most cases a service provider will bring a complete team. At least one member of the team must be committer of the corresponding eclipse projects.

3.4. Architecture Committee & Quality Committee

The committees elaborate standards and guidelines the developers commit themselves to adhere to. The committees reflect about and revise their regulations in regular intervals. They are responsible to define **HOW** the product is to be build and thus support the product owner in the definition of non-functional requirements.

3.5. Test team

A designated test team verifies that the implementation meets the defined acceptance criteria and does not breach any regulations. This test team does **not** replace the automated unit, integration and functional tests implemented by the development team.

Discussion 8-12-2014: It became obvious that this part of independent testing was neglected so far. The discussion led again to the importance of defined and automated test compatibility kits (TCK). We established, that there need to be at least two kits: . A functional TCK, specific for each component to test the functionality (WHAT) of a specified component. . A general TCK that is applied to all components to assure that the non-functional requirements (HOW) of architecture and quality are met. Later, Sven Wittig suggested that the test team should consists of the members of the quality

service.

3.6. Summary



Figure 1. The participants and their responsibilities

Chapter 4. Requirements

We suggest a light weight way to manage requirements using a backlog. A backlog is place where requirements are collected, grouped, sorted and prioritized due to business value (for further explanation of the term backlog, see http://en.wikipedia.org/wiki/Scrum_%28software_development%29#Product_backlog). We suggest to use a sophisticated issue tracking tool (like JIRA or YouTrack) to use as backlog as they offer advanced support by managing the backlog (filters, planning, different issue types etc.)

To support a light weight requirements process we propose the following issue types (issue == a general entry in the backlog):

- Epic: a big user story, feature. Takes certainly more than one iteration to implement
- Story: a user story, broken down so that it is possible to implement in one sprint
- Improvement: An improvement (change) to an existing story/task
- Bug: a problem which exists in the current code base and impairs the function of the product.
- Task: a task that simply needs to be done, not connected to a user story or epic.
- Subtask: a sub task of any issue

By using Epic and Stories as basic form to capture requirements, we accept that requirements are also subject to change. Therefore it is important to choose the right granularity for each context. After creation, the requirement should provide enough information to be understood by others and to be packaged by the PO (regardless if this is a person or the requirement management service) and prioritized by the SC. Of course to make estimations about the effort of implementation mostly more information is needed. We suggest that by creating a requirement the reporter (creator) agrees to deliver as much as needed initially (see template), but also agrees to be available for further detail specification and clarification once the requirement has been chosen to be implemented.

Since tasks are mostly so low level, that they are clear, we provide templates for epic and story, as well as bugs only.

4.1. Requirement template for Epic/Story

In order to ensure a smooth realization the requirements should already provide as many details as possible right from the beginning. At least the following information should be given:

- Summary

A short and yet precise summary eases the search for the requirements as it spares navigation effort in the issue tracking system.

- Description

The description should express which user group acting in which role (rights management) shall be enabled to execute which actions on the component. In short, the description should stress the benefit (business value) of the requirement rather than telling how sth should be implemented.

Proven format of user stories in different context are the following (but other forms as goal directed use cases etc. are also ok).

```
In order to [benefit]
as a [role]
I want [feature]
```

or

```
As a [role]
I want [feature]
So that [benefit]
```

To make a requirement ready to estimate, further detail information is needed. In agile project, defining acceptance criteria is extremely helpful for estimating. Acceptance criteria also help to define and narrow the scope of the requirement and support testing. Using examples to describe acceptance criteria is very useful, because they are easily understandable also by non-experts and allow quickly to gain a shared understanding. Acceptance criteria written in the following form are easily testable - with the right tools, an automated living documentation of what the product/component does can be built. This also eases the review and approval process of a component.

- Acceptance Criteria: (presented as Scenarios)

```
Scenario 1: Title
Given [context]
  And [some more context]...
When [event]
Then [outcome]
  And [another outcome]...

Scenario 2: ...
```

Of course this description can be enhanced anytime by a specific specification document, ui mock ups, sketches or whatever else is needed to implement the story correctly. These documents are attached to the main story/epic.

TIP

These scenarios can be used for acceptance testing by using a BDD (Behaviour driven development) like cucumber (<https://cukes.info/>).

4.2. Bug Reporting

Given that a flaw (bug) in the software is reported, then the explicit steps to reproduce it should be mentioned along with the actual and the expected behaviour. Also the version of software the bug was found in has to be specified. The more precise this description the easier the developers can advance to the root cause of the failure. If an error message had been displayed it is strongly recommended to attach its content to the bug description. In some cases screenshots of the failing system are of great help.

Chapter 5. The life of a requirement

This section describes how a requirement progresses through several states from its creation to the incorporation into the product increment. **New** requirements, feature requests or bugs can be filed by **any person** logged in into Jira. This of course involves developers who are adding functionality to the software and see needs for changes as well as end-users from inside and outside the working-group are welcome to report their requirements and to inform about malfunctions or unexpected behaviour they experienced while using the application. Additionally the quality assurance/test team can file deficiencies they discover while testing the application. This is actually true for both ways.

NOTE

As mentioned before, the PO/RSO is responsible to evaluate and classify these new requests. As she is also responsible for prioritizing the present requirements the reporter has no influence on the realization order. Priority or severity flags set by the reporter are only considered as recommendations.

At first the requests are added to the *Global Backlog*, a free accessible reservoir for all requests (in the state NEW, see workflow below). As the authors differ some of the requests might already describe in great detail what kind of addendum or fix is desired whereas others lack precision or just state a more general need. While different granularity is possible, they need to be at least ready for prioritization in order not to be rejected.

The PO/RSO is in charge to manage the backlog and hence to review all requirements. The review does not necessarily be done by the product owner himself. He may ask a developer for example to confirm a reported bug. Duplicate requests and those decided not to be implemented are **closed** immediately. Requests which are already implemented, invalid or just cannot be reproduced are declared **resolved** indicating the corresponding reason (e.g. fixed, invalid, worksForMe). The review can be seen as a constant clean-up task for the global backlog.

At this point, there is a difference on how the implementation is continued.

5.1. Life cycle in sponsored projects

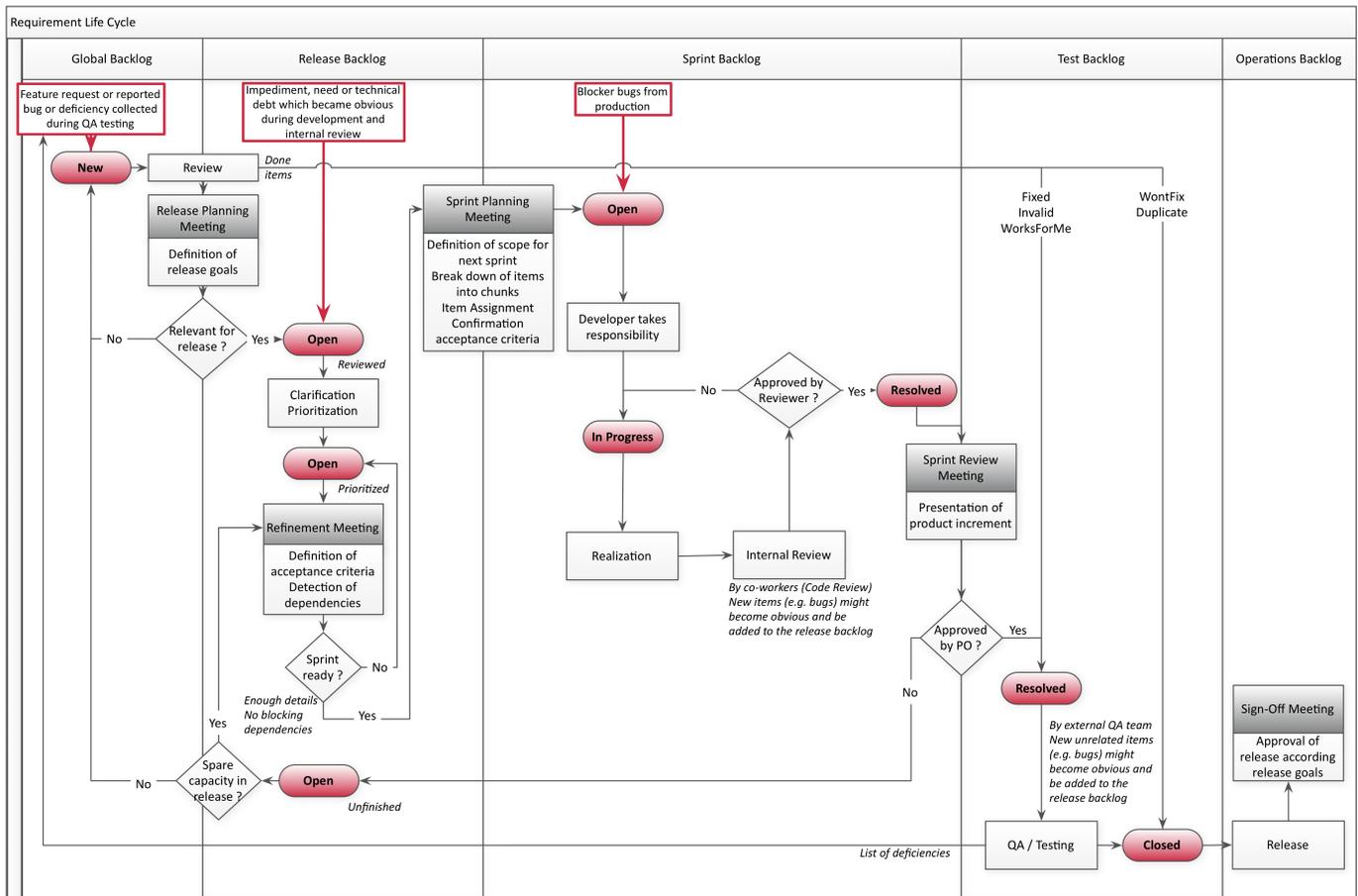


Figure 2. The requirement life cycle in sponsored projects

5.1.1. Issue Creation

In addition to issues created by anybody, high level requirements or work packages can be offered or suggested to the steering committee. These are managed separately on the openMDM website. They get an id and thus can be traced also later in the requirement/issue tracking tool. The steering committee (SC) of the OpenMDM working group and the designated PO or RSO regularly come together to a *Release Planning Meeting* in order to discuss and define the overall, coarse-grained release goals for the next product release. Based on the agreements of this meeting the PO/RSO is then responsible for evaluating if a certain requirement is relevant for the upcoming release. If the issue is of relevance for the next software release then it is moved to the next *Release Backlog* (the Release Backlog is more a filter of the global backlog, not another physical repository) and its state switched to **open**.

5.1.2. Detail Specification, Clarification and Planning

The following section describe a proposed process for sponsored development, but of course the sponsor and the provider are free to organize themselves as long as the workflow and the AC and QC guidelines are followed.

Once a requirement is selected into the release backlog the requirements have to be made ready for estimation as quickly as possible by adding acceptance criteria and other detail specification (mock ups etc.). The product owner contacts the initiator (reporter) of the requirement request in order to fix

its content and scope. Furthermore the PO has to confer with the steering committee to clarify the relevance of the requirement from the business's point of view. In addition to that the product owner prioritizes the backlog items according to their importance to release success.

In regular *Refinement Meetings* the product owner and the developer team inspect the requirements. The first goal of the meetings is to establish a common understanding of the tasks at hand. Furthermore acceptance criteria need to be reviewed and complemented where needed. Last but not least impediments such as dependencies between items need to be identified. Thereby quality guidelines and architectural standards have to be considered. The developers estimate the realization effort i.e. the costs for each item. The farther the project advances i.e. the more experience the team gains and the more often an item is discussed the faster the estimation will narrow down to a reliable value.

NOTE Nonetheless estimates are no 'exactimates'. By definition they are prone to some degree of inaccuracy.

NOTE In the following paragraphs we assume an iterative or flow based agile approach is followed. However, as stated in the discussion from 8-12-2014, it is open to the sponsor and the development team how they are going to implement the issues in the release backlog. Nevertheless regarding the involvement of many players and stakeholders we strongly recommend following an agile approach, which includes frequent integration of product increment. This ensures that learning happens as early as possible. **The later the integration of different parts and components, the later the learning the more expensive the corrective measures.**

5.1.3. Ready for development (aka sprint ready)

We recommend that content of the release backlog is realized in several iterations, also called **sprints** or several deliveries are made if using a flow system (Kanban). The term sprint is again derived from the Scrum framework and describes a fixed work phase. The items regarded as 'ready for development', i.e. detailed enough, provided with acceptance criteria and not blocked by any known issues constitute the basis for the *Planning Meeting* which is held at the beginning of each iteration/regularly. During this meeting the product owner and the developer team check the available manpower, define the scope of the next sprint, split large items into feasible sub tasks and finalize their acceptance criteria. Moved to the *Sprint Backlog* the issue remains **open** till one developer with free capacity signs responsible for it.

5.1.4. Realization and internal review

The requirements then enter the realization stage and are declared as being **in progress**.

The specified release backlog items are developed according to the architecture and quality requirements and guidelines. As shown in the workflow, a committer should review the implementation considering QC and AC guidelines as a first quality gate. The final approval is made by the QC and AC members (see also Approval & Testing)

When at some point the implementation gets blocked this immediately has to be communicated to the sponsor and the sponsor. The solution of these impediments takes priority. Another principle of agile development is that a story is either done or not done. So regardless the cause (impediments, wrong estimates), if a story is not finished, it moves back to the release backlog (and most commonly into the next sprint backlog). This is another reason to work with an experienced product owner. She is familiar with story splitting and the smaller the items the bigger the chance they get done!

When the implementation is considered finished it is set to implemented and waiting for review and to be committed to the source repository/merged to the master branch by a/another committer.

5.1.5. Approval & Testing

At the end of each sprint/iteration (but latest at the release if not followed an agile approach), the product owner, as representative of the stakeholders, approves the implementation and test result during a *Sprint Review Meeting*. Together with Architecture and Quality Committee she checks that the software increment fulfills the functional and non-functional demands. If the product owner is not satisfied with the provided solution then the task is **reopened** and put back into the release backlog.

NOTE

This does not prevent the product owner from verifying resolved items prior to the sprint review meeting. If she does not consent a certain implementation and the developer team has capacity left in the current sprint then she may reopen the issue and keep it in the sprint backlog. A developer, most preferably the developer who was responsible for the implementation in the first place (as she possesses most knowledge about it) will take care of it. A reopened task which cannot be fixed within the ongoing sprint is forwarded to the release backlog.

The product increment and the realized items are eventually handed over to the quality assurance and testing department. The testers check once again that all acceptance criteria are met and verifies that the implementation follows the agreed quality guidelines and architectural standards. The test team expresses its accordance by setting the item's status to **closed**. In case deficiencies are diagnosed then **new** issues are added to the global backlog.

NOTE

The advantage of involving the test team after the product owner has signed off the result of the sprint lies in the fact that the testers can work at their own pace (decoupling of work). They are not forced at the end of a sprint to rush through testing to keep the Sprint Review meeting deadline.

After the last sprint of a release has been concluded all then closed issues are integrated in a concrete release deliverable. This deliverable is presented by the product owner to the sponsor(s) of the release and to the steering committee in a *Sign-Off Meeting*. The stakeholders, considering the formerly defined release goals, accept or deny the given solution. Only in the former case the solution is put into production. Either way the next iteration starts with another *Release Planning Meeting*.

5.1.6. Debts of former sprints

As long as the last sprint for the current release scope has not been started unfinished tasks can be

considered in the *Refinement Meetings* for the next iterations. It is the product owner's responsibility to sort the tasks into the prioritized list of backlog items. In case the last sprint of the release is already fixed then the task is, along with all remaining release backlog tasks, moved to the *global backlog* and considered again as **new**.

5.1.7. Final remarks

The clarification meetings have to take place regularly. Items can only be considered for a sprint planning meeting when they are clarified, described in detail and prioritized. It is crucial that the product owner prepares the designated next sprint's items while the current sprint is still ongoing.

Lessons learned will be gathered in *Sprint Retrospectives* at the end of each sprint which may influence the realization process of upcoming sprints.

At any time, if an exceptional condition arises the product owner is allowed to notify the steering committee. In the following a non-exhaustive list of such conditions is provided:

- a high priority ticket cannot be solved in time
- an impediment requires the decision of the steering committee
- a new highly valuable capability has been discovered that could be turned into a requirement.

In case a blocker issue is diagnosed in the productive system, then it is added to the current sprint backlog at once. The developer team and the product owner have to negotiate which other items are then dropped from the current sprint backlog and postponed till the next sprint.

5.2. The classic open source life cycle

Volunteer contributions are also handled the normal open source way - contributors contact a committer or the requirements service owner to tell them, that they are interested in implementing an issue, then the volunteer contributor submits a patch and the committers decide if the code is included (using the AC and QC guidelines). Finally a member of the QC checks these guidelines again and accepts the contribution by closing it.

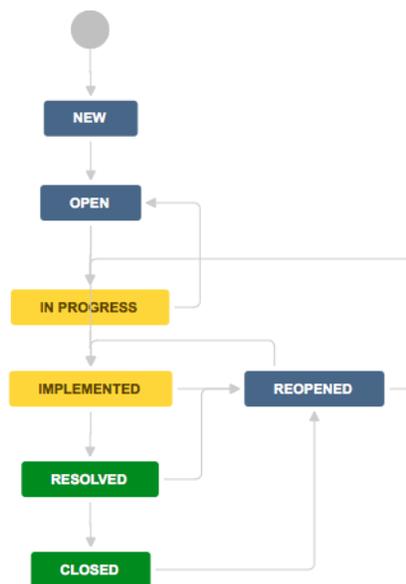
Chapter 6. Requirement/Issue Management Tool (Jira)

The working group decided to use Jira as the requirement and issue management tool. A Jira on the atlassian cloud has been installed: <https://openmdm.atlassian.net/>. There is one project OpenMDM Backlog (OMDM) which is the central place to catch the requirements. The issue types described above are available. High level requirements from the openMDM website are mapped to epics using the id in a custom field.

NOTE When using JIRA for development, we must link the sprint reports to bugzilla so that bugzilla users get an overview of the current state of the project. (example: https://bugs.eclipse.org/bugs/show_bug.cgi?id=467712).

6.1. Workflows

The above life cycle of a requirement is represented through the OpenMDM workflow:



The states have the following meaning:

- NEW: The issue has been created and needs to be evaluated (duplicate, etc.).
- OPEN: The issue has been selected for implementation

- **IN PROGRESS:** The issue is being implemented
- **IMPLEMENTED:** The issue is considered implemented by the team and ready for review/merge into eclipse
- **RESOLVED:** The issue has been reviewed by a committer and if ok, is integrated into the eclipse repository.
- **CLOSED:** The issue has been signed off. This usually includes a review of sponsor, architecture board and quality board.

Tasks and subtasks do not have the status NEW as they are set to OPEN when they are created. This reflects the typical usage of tasks and subtasks as they are created by the development team during sprint planning. For this reason there is a special task workflow to reflect this.

6.2. Security

Jira offers in terms of security and permissions the concepts of groups and project roles (see <https://confluence.atlassian.com/display/JIRA/Managing+Project+Roles>). Project roles are - in comparison to groups - project specific and by default the project permissions (who can create issues etc.) are based on project roles. As there will be only one Jira project reflecting the various EWG projects and to keep the Jira configuration simple the default project roles (Administrators, Developers and Users) are used to manage all permissions except issue transitions. Those are based on group membership as they need to be more fine grained.

The organisational roles are mapped to Jira groups as follows

Table 1. Organisational Role - Jira Group - Mapping

Organisational Role	Jira Group
Release Owner (SC Member, Patron, Delegate of Patron)	release owner
Developer/Team	jira-developers
Committer	openMDM committer
Architecture & Quality Committee	architecture/quality committee
Logged in User	jira-users
Requirements Service	administrators + site-admins

The following table shows the organisational roles and their project role memberships

Table 2. Organization Roles (Jira Groups) and Jira Project Roles

Jira Project Role	Release Owner (SC Member, Sponsor, Delegate of Sponsor, PO)	Developer/Team	Committer	Architecture & Quality Committee	Logged in User	Requirements Service Owner (RSO)
User	x	x	x	x	x	-
Developer	x	x	x	x	-	-
Administrator	-	-	-	-	-	x

The RSO, i.e. the representing Jira user, is the only one having administration rights in Jira. Due to the technical importance of this role a substitute should be named.

The following table shows the organisational roles and which issue transitions they are allowed to perform

Table 3. Organization Roles (Jira Groups) and Workflow Transitions

Transition	Release Owner (SC Member, Sponsor, Delegate of Sponsor, PO)	Developer/Team	Committer	Architecture & Quality Committee	Logged in User
Creation (NEW/OPEN)	x	x	x	x	x
NEW - OPEN	x	x	x	-	-
OPEN - IMPLEMENTED	x	x	x	-	-
IMPLEMENTED - RESOLVED	x	-	x	-	-
RESOLVED - CLOSED	x	-	-	x	-

Though it would be good to restrict the creation of tasks and subtasks to everybody but users, Jira is not capable of adding permission schemes to workflows and/or groups.

6.3. References

To reference the requirements of the Eclipse Project, there is a Jira custom field "Requirement ID", a label field, allowing to add a free text reference information. To maintain the assignment of the issues to the corresponding Eclipse project the Jira custom field of type select list offers an optional selection of available Eclipse projects to reference. This list has to be maintained by the Requirements Service. OpenMDM components (according to the architecture document) are visualized in Jira as Jira components (e.g. Navigator etc.)

6.4. Other Jira Projects

To manage the tasks of the different committees another Jira Project (OpenMDM 5 Organisation) has been created (ORGA). To make filtering between the different committees easier, a custom field of type select list with the committees as entries has been created. The workflow of this project is much easier, there are just three states (TODO, In PROGRESS and DONE). The available issue types are tasks and subtasks, request for comments and request for decision.

Chapter 7. Direct comparison of two organisational approaches regarding requirement management

To make the discussion/decision easier, the initial suggestion by Canoo and alternative suggestion by Sven Wittig are presented below with a short summary and a SWOT analysis.

7.1. Suggestion by the requirements management process service (Canoo)

7.1.1. Summary

1. dedicated, independent product manager (Scrum role product owner) which is reporting to the steering committee
2. product manager coordinates all planned (top down) development around openMDM. S/he has know-how of lean product management and can support service provider implementing the product in iterations - product manager is involved in development (no blackbox). Of course the product manager can delegate some of the daily work, but remains responsible.
3. Development which is contributed by the community without having been planned by the work group is of course possible and welcome, the contributions are evaluated by the test team and the product manager

The process is visualized in the following image:

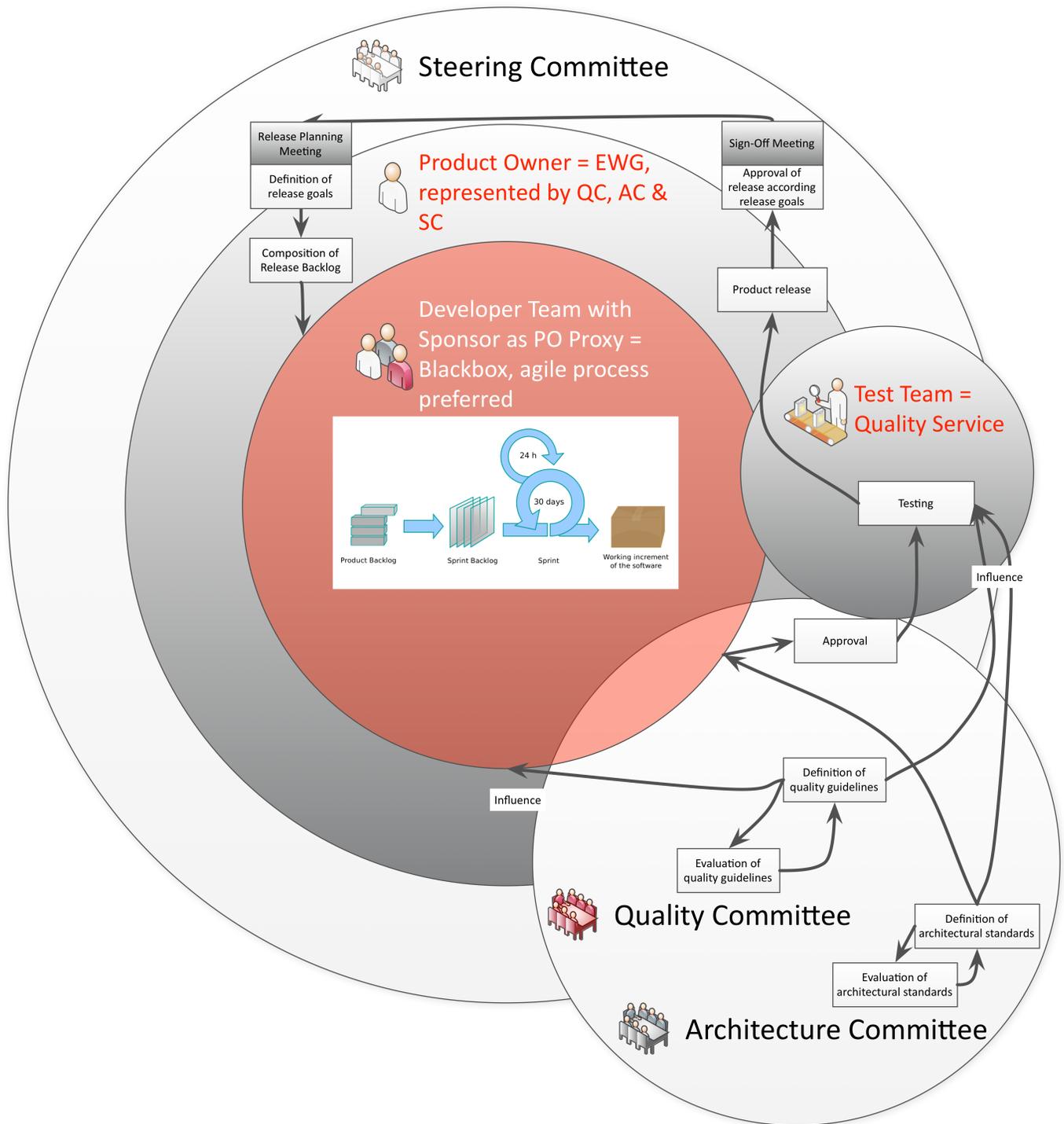
<p>Strength: focused and lean product development possible - central roadmap and release planning help to lower time to market. High quality of comprehensive product (building blocks and example system) less overhead, rework because of short feedback/decisions cycles.</p>	<p>Weakness: full time job - increases cost for work group</p>
<p>Opportunities: shorter time to market of a comprehensive work product. This increases adoption of the product (aka Baukasten) of the automotive community, which then increases possible contributions (viral growth)</p>	<p>Threats: too much knowledge concentration at the product manager. product manager is not really independent.</p>

7.2. Suggestion by Sven Wittig

7.2.1. Summary:

1. No dedicated, work group Product Owner, but product ownership is executed by whole EWG, represented by QC, AC and SC
2. The quality service execute the tasks of the test team
3. Black box of implementation: Service Provider takes the requirements (functional and non-functional which include quality and architecture guidelines), discuss everything internally with its sponsor and delivers the implemented results back to the workgroup. They are totally free which methodology to choose, the only constraint is that the issues are publicly available in the issue tracker
4. The acceptance of the delivery is decided by the sponsor and by the Quality Service (test team).
5. The requirements service is limited to collect and organize incoming requirements from the community, bundle them and present them to the work group.

The following picture should visualize the suggestion



Strength:

less coordination work which also means less costs for the work group
 only built what finds a sponsor

Weakness:

long time until decisions are made because of common product ownership
 lack of focus: danger of implementing similar components slightly different instead of focusing of providing the main components as building blocks

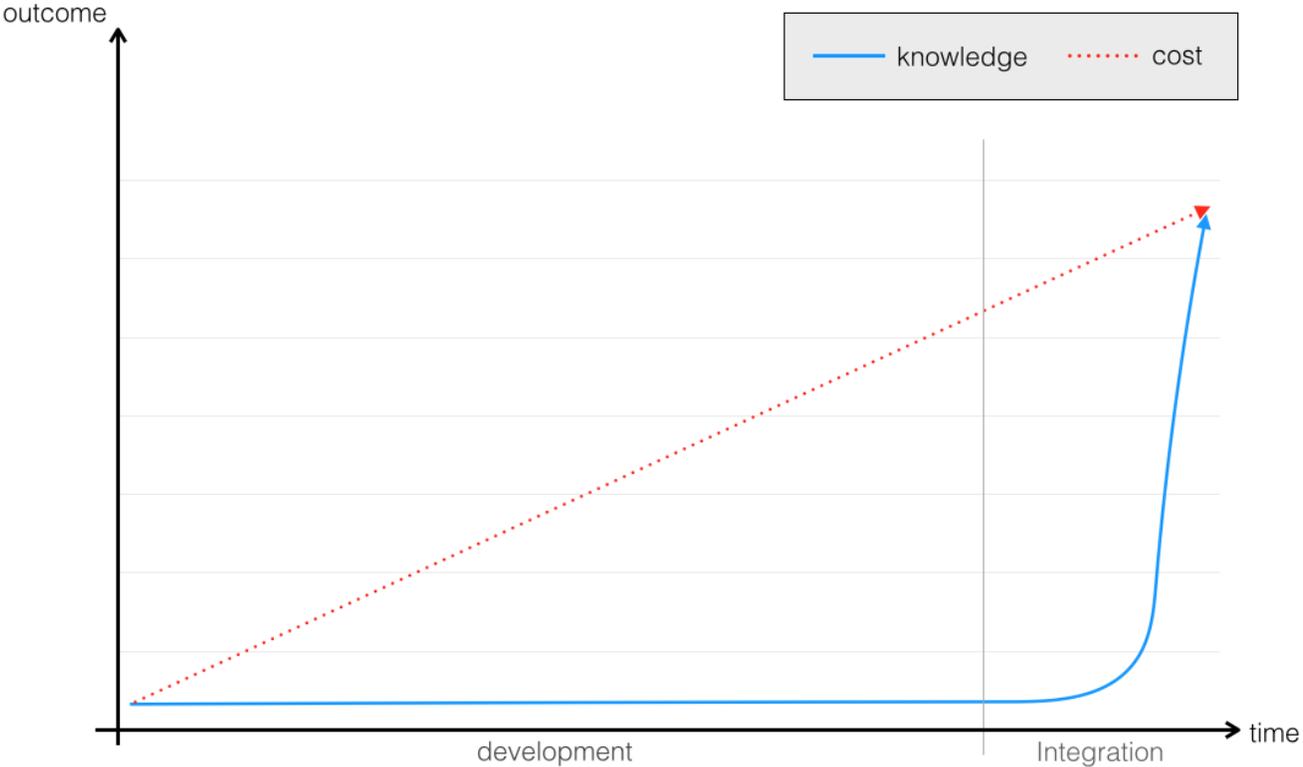
<p>Opportunities: self-driving community emerging components which nobody thought of beforehand</p>	<p>Threats: lack of coherence of the individual building blocks (despite QA) individual interests of the sponsors are put above the interests of the work group. longer time to market prevents community adaption ("we rather build our own thing now")</p>
--	--

7.2.2. Conclusion

Both models have their advantages and disadvantages.

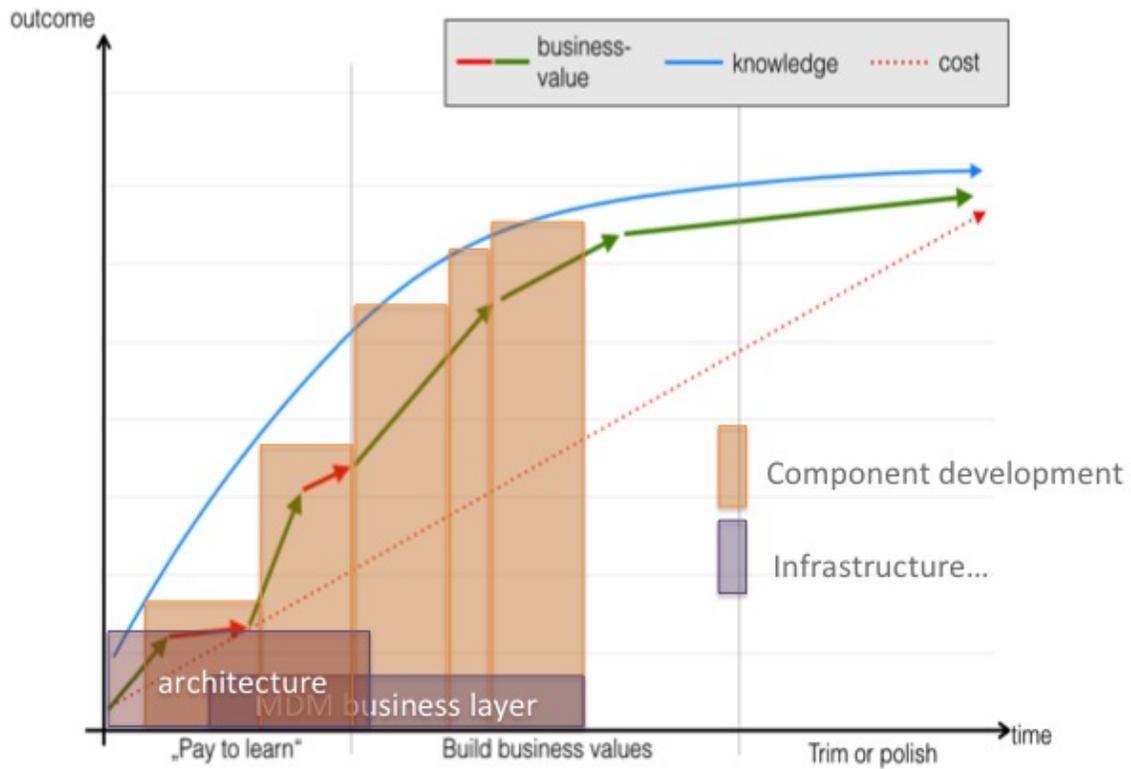
With the current situation, where orders are more "horizontally" organized (MDM API, Web client, Rich client) etc. we strongly suggest the model with an overall product responsible which coordinates the different orders. Why is it important to coordinate:

If separate packages are delivered and integrated at a very late state, not only will any learning happen very late (often too late for the project to be successful) but there is also no real value to the end user delivered until very late as shown in the following picture:



In order to get value earlier, strong coordination in terms of functionality between the individual work packages is required. Hence the idea of an overall product owner, which makes sure that in regular intervals working software is provided. S/he is also responsible to drive the implementation with requirements.

If on contrary "vertical" self-contained work packages are ordered, every delivery results in a usable, working product (which of course has to be compliant to the architecture). With this approach it is sufficient to coordinate in the steering committee because there is not much dependency between the different work packages.



Because time to market seems essential for the success of OpenMDM 5 and taking the current situation into account, where everybody seems to be waiting for anybody else, we strongly suggest to coordinate the efforts of the already started work packages. In the mean time when creating new work packages, they should be self contained and deliver business value, so that they can be managed independent of each other.