An Approach for Requirements Engineering for Software Library-Components and Patterns to be Reused in and across Product Lines

Dirk Herrmann
Robert Bosch GmbH
Robert-Bosch-Allee 1, 74232 Abstatt
Dirk.Herrmann2@de.bosch.com

Dr. Jens Liebehenschel
metio GmbH
Sulzbacher Straße 105, 65835 Liederbach
Jens.Liebehenschel@metio.de

Abstract

In the development of product lines one of the challenges is to exploit commonalities among products such that development cost is reduced. For software, this implies to identify parts that are sufficiently similar to define library-components or patterns for reuse. The detailed definition of scope, responsibilities and interfaces for library-components and patterns is part of their design process, which makes requirements engineering different for them.

The paper presents a method to determine a set of requirements that forms a suitable starting point for the design. This assures that the component or pattern to be designed fits well to all systems under consideration. Established design principles like abstraction or parameterization are applied to requirement engineering and management. The method has been successfully applied in several automotive cross-product-line development projects.

Motivation and Introduction

Solid requirement engineering is the base for successful delivery of products in time, with the desired quality and within the cost limit [1, 2]. Good requirements are necessary for appropriate designs and reliable and complete tests. The reason for about 40-50% of the problems (cost or time overruns and deficiencies) in software projects is closely related to requirements [3, 4, 5] – 15 years ago and still today. So, effective requirement engineering and management techniques are crucial for the success of projects, and they belong to the biggest challenges [6].

In general there is more than one requirement engineering phase during the development of systems. For example according to Automotive SPICE [7] there is the phase of elicitation and analysis of the system requirements as well as the phase of software requirements analysis. Often – especially in larger projects – there are additional phases of requirements analysis, e. g. for system or software components. Another additional phase of requirements engineering is necessary in the development of product lines, the scoping [8, 9, 10, 11], targeting top-level requirements.

The paper is about handling of requirements for components and patterns that shall be used in different contexts. It focuses on stakeholder communication and the documentation of the results of one requirement engineering phase. In particular we look at components or patterns for which the detailed scope yet has to be decided and for which therefore also the external interface is subject to design considerations (in contrast to standard components as defined by AUTOSAR [12]).

One of the key points is that commonalities and variability have to be managed efficiently. This holds for product line development, but even more for component development for different product lines or products. The fact that we are concentrating on only one phase is no restriction for general applicability, since all requirement engineering and management phases of a project can be combined in a natural way for the sake of obtaining the overall requirements traceability demanded by [7, 13].

Our approach was applied successfully in several cross-product-line development projects, like replacing a set of rivaling component implementations by a common solution, extending a component's scope of reuse to a new context, developing architecture level design patterns, redesigning an existing library and developing a new software component.

The term *requirement* in this paper is meant to cover requirements of all kinds: functional as well as non-functional, qualitative as well as quantitative, problem space focused as well as solution space focused, externally observable or not.

Consequences if scope and interfaces are subject to design decisions

Assume that embedded software for a number of products has to be developed, and it is observed that there is some commonality between the products. For example, the designers might realize that there is the need for some kind of sorting capability to fulfill the requirements of several products. In such a scenario, in order to improve development efficiency, it appears plausible to investigate the possibility of developing a sorting library that is to be reused between the products.

What are the requirements on such a sorting library? This question is difficult to answer at the described stage of development:

- It has yet to be decided whether developing a common library is more economic than if each of the different products implements its own solution for its own specific needs. If that option was taken, there were no requirements on a sorting library at all.
- If only one of the projects requires a stable sort: Is it a requirement on the library that the sort algorithm shall be stable? (A sorting algorithm is called *stable* if the relative order of records with equal keys is preserved.) If it is at this stage still a design option that the library is designed to only fulfill the needs of the majority of projects, then the resulting sort algorithm is allowed to be unstable. Therefore, it cannot in the strict sense be a requirement for the library's sort algorithm to be stable.
- On the other hand, if making the sort algorithm stable will not cost more (or only little more), it may still be decided to support the one additional project by providing a stable sort.

In other words, there is no defined set of requirements on the sort library at first. In fact, defining the scope and interface of the library is a design step that yet has to be taken, and only afterwards the set of requirements is known. During design of the library, the designer will strive for a solution that will be most efficient across the set of projects. Thus, the designer will need an overview over those requirements from potential client projects that could have an impact on the design of the sorting library.

To generalise from the sorting library example: For the design of assets like libraries, re-usable components and patterns, there is no set of specific requirements at first, but this has yet to be defined as part of the library design. However, there will be generic non-functional requirements related to the business goals, like:

> *Define the library such that the combined effort of library development, library integration and project specific implementations for the set of projects under consideration is as close to the economical optimum as you can get.*

Superset of requirements

The goal of requirement elicitation for a system is, even if seldom reached in practice, to collect all requirements that are relevant for the design of the system, ideally before the design activity starts. Such a set of requirements shall have no contradictions and it shall contain no requirements for which there is no actual current demand.

Our approach, in contrast, focuses on the development of assets that shall be usable in different contexts. Consequently, central to our approach is the definition of a *superset of requirements* that contains all requirements from potential client projects that could have an impact on the design of the asset. The requirements in the superset do not necessarily have to be fulfilled by the re-usable asset. They do have to be fullfilled by the clients, possibly by making use of that asset.

Due to the different targeted contexts, the resulting superset of requirements will have the following properties:

- Some requirements are only relevant for some of the contexts. For the sorting library example, depending on the context it may be required to sort in increasing order or in decreasing order. Some contexts may even require being able to sort in both ways.
- There can be requirements from different contexts that, from an abstract point of view, demand the same property from the system, but to different degrees. The maximal response time for example can be different in the different contexts.
- Requirements that stem from different contexts can even be contradictive. For some contexts it may be required, for field return analysis purposes, to provide access to certain internals of the system, whereas in other contexts, for security reasons, it may be required that the access to the same internals is practically impossible.

One source of requirements were iterative sessions with different stakeholders from the different contexts: There were several sessions with stakeholders from the same project, interleaved with sessions with stakeholders from other contexts. Where previous solutions existed there were also sessions with the respective developers.

Further sources of requirements were documents and existing code, which were analysed offline before or between the sessions. For some components we even created functional models in a high level scripting language. This helped to identify open questions that otherwise would have arisen during the later design phase, and thus helped to clarify some of the requirements early. As we became better acquainted with the problem, we also added requirements to the superset that might have been missed so far and put them up for discussion.

For the full benefit from the *superset of requirements*-strategy we also added requirements that

- were not demanded yet by any of the stakeholders, but might be demanded in the future within the known or additional contexts,
- might be allocated to the system under consideration by a design decision, or
- were included only to state explicitly that they are not required.

Obviously, this approach lead to a larger number of requirements and demanded a clear structuring.

How this was achieved will be discussed below. However, the approach had the following benefits:

- Knowledge pool: We began to treat the resulting set of requirements as a knowledge pool that would allow a long-term use, also in case of changes to the set of clients or client's requirements.
- Design for future extensions: Since we also added requirements that were not demanded yet but were likely to be required in the future, the system could be designed to be easily modifiable in these directions later.
- Know-how transfer between clients: Due to the interleaved requirement elicitation sessions the different stakeholders learned from each other. E. g., requirements which were added in one session as potential future extensions were later identified as actually required by another client.
- Design freedom: Even requirements that were only remotely related to the asset under consideration were added, leaving decisions about the scope of the library to the design phase. For example, the superset of requirements for an EEPROM subsystem contained the requirement of some projects to store encrypted data, thus leaving open whether the EEPROM subsystem would be responsible for the encryption.
- Avoiding misconceptions of clients: The clients would implicitly expect certain requirements to be fulfilled, which, for example due to a principal technical impossibility, would not be fulfilled. These were added and marked as not fulfilled during the respective session, in order to document the shared view of the situation. For instance, the requirement that an error detection code shall detect every possible data corruption scenario is technically not achievable.
- Avoiding misconceptions of designers: Similarly, the system designer might have some requirements in mind which would actually not have to be fulfilled. Those we became aware of were added and explicitly marked as not required. For example, a designer might assume that a certain software component would have to be reentrant or would have to work in a system with a non-preemptive scheduler.

Freedom for the design phase

The requirements document is the most important input for the design process. Designers shall not be lead to misconceptions about what the clients really need. They shall also not be unnecessarily limited with respect to the solution space. Therefore, for each requirement in the superset the relevance for each client was remembered in detail. For example, if several clients had different demands on a system's response time, every client's demand was noted instead of simply noting the toughest demand. Other information was added as well, like if a requirement was added because it seemed likely that it would be demanded in the future.

This detailed knowledge allowed the designer to choose between several ways to deal with different requirements for different clients.

- It could be decided to provide individual asset realisations for certain clients, e. g. if the usability by these clients would not outweigh the resulting disadvantages for the other clients.
- Another option is to provide a set of library elements which allow to be integrated differently by each client. Such a library would provide optional elements for optional features and alternative elements for contradicting requirements.
- Assets that can be designed such that, instead of integrating different elements from a library, clients will always integrate the same elements, but have to configure them according to their needs.
- In case of non-contradictive requirements, a single ready-to-use asset can be provided that fulfills all requirements - even for those clients, where they are not needed. This avoids system variants and can be beneficial if the fulfillment of the additional or tougher requirements is cheaply achievable.
- And, certainly, the above approches can be combined, for example by providing a set of library elements, some of which need to be configured, some of which fulfill simply the toughest requirement, and some of which are designed to be replaceable by context specific individual components.

Requirement	**Client A**	**Client B**
The component shall be usable in an environment without preemptive scheduling.	Yes	no
If any input signal is unavailable, the component shall not influence the actuators.	Yes	Yes, but also if signal x is unavailable
If any input signal is unavailable, the component shall make this visible to the error memory unit.	Yes	

We chose a matrix form (as shown in the example above) to indicate which requirement applied to which client, since this provided a much better overview than, for example, different files for different clients or separate entries in a data base.

This table was created in the iterative client sessions that are described in the previous section. Blank fields indicated that the client was either unsure or not yet queried about the respective requirement. Moreover, as shown in the example, we always allowed the clients to add comments rather than demand them to stick to yes/no answers. These comments helped us to rework the set of requirements for the subsequent sessions and iterations.

Parameterized requirements

Our approach not only demanded to manage a larger set of requirements, but combined this with the intent to record for each requirement in which contexts which requirement had to be fulfilled. One strategy we used in order to make this manageable was to introduce the concept of parameterized requirements.

A *parameterized requirement* is a template text with one or more placeholders, from which a set of requirements can be derived by substitution of the placeholders.

Requirement	Client A	Client B
The function shall be usable within {left hand drive cars, right hand drive cars, both}.	Both	Left hand drive cars
The driver's side shall be determined at the time of {build, system manufacturing, car manufacturing, system boot}.	System boot time	Build time
The response time of the function shall be below {time}.	below 500ms, but not smaller than 100ms	250ms

The example shows some possibilities for the parameterization of requirements: The substitution value of the placeholder may have to be chosen from a limited set of options to choose from, or it may be a numeric value.

The process of developing the parameterized requirements was iterative. After some clients had provided requirements we identified possibilities for abstraction. For later iterations with the same clients as well as with additional clients the already parameterized requirements were presented. This allowed us to improve on the formulation of the requirements and the set of possible substitutions. Again, as shown in the example, we always allowed the clients to give more details rather than demand them to stick to the suggested options.

The concept of parameterized requirements was a powerful mechanism to eliminate redundancy and to create abstractions for sets of similar requirements without loss of information and thus without restricting the design space. And, as can be seen from the example matrix, it provided an excellent overview over the variance for the respective requirements.

Asset specific usage profile forms

For the design of an EEPROM subsystem it is helpful to know for a client project the set of data elements to be stored, their respective sizes, the events in which they should be written, the effect in case of data loss, the effect in case of undetected data corruption and many more. These *usage profiles* are likely to be different for different clients.

Formulating this information in form of requirements is awkward: Treating each detail as an individual requirement leads to a large number of requirements, many of which are subject to change during a typical project. Subsuming the information like only naming the maximum size of a data element and the total number of elements is misleading and can lead to inappropriate design decisions. For example, the knowledge that there is likely to be a large number of one-byte data elements to be stored on the EEPROM will make the designer think twice whether it is a good idea to give every data element its own 16-byte EEPROM sections.

We created for each asset a usage profile form, in which the usage information specific for that particular asset could be entered. For every asset the profile information had a different structure: The usage profile form for the EEPROM subsystem was different from that for the software timers. For each client project of an asset, one instance of the respective form was filled out.

Similar to the way parameterizable requirements were identified, the definition of the usage profile forms was an iterative process that made use of requirements or other information gathered so far. For example, for the EEPROM subsystem clients would state that "some of the data elements are absolutely essential for the operation of the ECU". This indicated the requirement to provide special availability for selected data elements. But in fact it indicated the more general concept that different data elements might have different availability requirements. Thus, the usage profile form was extended by a required degree of availability to be given for each data element.

Usage profile forms were not filled out by all projects, and even for those projects that provided them, updates were only made infrequently. That meant, that at any time the information in that forms was likely to be inaccurate. Still, the overview provided by these forms was very valuable information for the design. The inaccuracies never lead to a problem in practice: First, the designers were aware that the information was subject to changes in many details. Second, focusing too closely on a particular usage profile would have reduced the changes for future re-usability.

Constraints versus solution ideas

The superset of requirements approach implied that formulations, which were solutions rather than requirements, were also noted. We observed that there were two reasons why the stakeholders formulated solutions as if they were requirements:

- They were used to talk about features from the solution space rather than about the underlying requirements. However, they signaled willingness to accept other solutions after the underlying requirements were identified. In these situations the solution was not meant as a constraint but rather as a suggestion.
- Certain solutions were explicitly demanded, despite knowing that alternative solutions might be possible. Or, some solutions were explicitly excluded. A common reason for these situations was bad experience with other solutions on the customer side. Thus, in such cases the solutions were constraints.

In both cases we worked out the underlying requirements (either offline or directly with the client) and added them to the superset of requirements for discussion in subsequent sessions.

Solutions provided as suggestions were not kept as requirement entries of their own, but rather were added as supplementary information titled 'solution ideas' to the underlying requirements. We primarily did this for those solution ideas that were provided by the stakeholders, since it was not our intention to turn the document into a design document.

However, we learned that often the stakeholders still preferred to talk about the requirements in terms of the solutions they originally had in mind. Thus, adding the solution ideas to the underlying requirements made the communication with the stakeholders easier. It also avoided the impression that the concepts they were familiar with would not be considered during the design phase. And, finally, it documented the abstraction that was performed by elaborating the underlying requirements from the solution ideas.

In contrast, in those cases where the solutions were actually meant as constraints, we

- kept the solution as a requirement of its own,
- classified it as a constraint,
- marked it as required by the respective clients,
- added the reason for the constraint, and
- added references to the underlying requirements.

Further issues

Documentation of non-obvious cross references: Requirements that were referenced from within or from outside the document were given a symbolic name for identification. We chose symbolic identifiers over numbers due to their mnemonic advantages. We focused on references that documented non-obvious relationships. Therefore, for all such references the reasons for these references were given rather than just setting unexplained links.

Requirements documents and configuration management: The superset of requirements approach made it possible to keep information from several contexts in the same requirements document without need for complex configuration management processes: Changes to the files included corrections, performing abstractions, extensions and deletion of information that was not relevant any longer. All this could be achieved by versioning the requirements document, but without need for branching.

Separation of process: The requirement document was left clear from process issues like information for planning and tracking, dealing with change requests etc.. The respective information was placed in separate documents. This resulted in a looser coupling between the requirements engineering process and the other processes and thus allowed for the independent development of appropriate handling mechanisms.

Practical application and results

The method has been successfully applied for the development of several cross-product-line assets for automotive projects, like

- replacing a set of rivaling implementations by a common solution,
- extending a component's scope of reuse to a new context,
- developing design patterns,
- redesigning an existing library and
- developing a new software component.

The assets to be developed came from different levels of abstraction, namely

- software architecture level,
- sub-structured top level components and
- bottom level libraries.

The requirements elicitation sessions were performed with stakeholders from one to three projects that planned to make use of the yet to be developed assets. There were at most three sessions of about one or two hours with each project, often less, because in later sessions only few new requirements were added.

Between the sessions we analysed the provided documents, worked on the document structure, improved formulations, performed parameterizations and identified constraints. We found that parameterization reduced the number of requirement lines in our documents by about 25%.

Maybe surprisingly, after the assets had been implemented we only very rarely were faced with new requirements, both from the originally targeted clients as well as from clients that later became interested in the assets. Most of the later clients did not even go through a thorough requirements elicitation phase, but instead based their decision to use the asset based on the asset's feature description.

One of the assets, shortly after being designed and implemented, was used in 17 different client projects belonging to seven different software product lines.

Summary

We have presented various aspects of a requirements engineering method that helped us to generate requirements documents that among other benefits

- was well suited for communication with the stakeholders and designers,
- provided an excellent overview of the common and variable requirements,
- avoided unnecessary restrictions of the design space,
- allowed for continued usage of the document,
- supported the transfer of know how between different stakeholders, and
- could be used as questionnaires during requirement elicitation sessions.

During the application of the method we applied principles like abstraction (e. g. usage profiles) and single source (e. g. parameterized requirements), separation of concerns (e. g. connection, but not integration of process issues in the requirements document), low coupling (limiting cross reference to non-obvious cases) or verbalization (e. g. usage of symbolic names for requirements in cross references) in order to eliminate redundancy, support the readability of the requirements document and to improve process efficiency.

The process was agile for some reasons: We continuously improved the structure and contents of the document as well as the details of our method. We valued content over form by encouraging clients to use their formulations rather than requesting them to stick to the predefined format of some parameterized requirement or usage profile form.

The core elements of our method were

- collecting a superset of requirements,
- parameterized requirements,
- usage profile forms filled out by each client, and
- remembering constraints as client provided solution ideas.

As we have applied this approach for several systems on different levels of abstraction, we are convinced that many of the above concepts can be applied to other kinds of projects and even in other domains.

References

[1] K. Pohl: Requirements Engineering: Grundlagen, Prinzipien,Techniken, 2nd ed., 2008
[2] C. Rupp et al.: Requirements-Engineering und -Management, 5th ed., 2009
[3] The Standish Group: CHAOS, 1995
[4] T. Hall et al.: Requirements Problems in Twelve Companies – An Empirical Analysis, Proc. of EASE 2002
[5] The Standish Group: CHAOS, 2009
[6] European Software Institute: European User Survey Analysis, Technical Report ESI-1996-TR95104, 1996
[7] Automotive SPICE, http://www.automotivespice.com/
[8] G. Böckle et al.: Software-Produktlinien, 2004
[9] P. Clements et al.: Software Product Lines, 2002
[10] F. van der Linden et al.: Software Product Lines in Action, 2007
[11] K. Pohl et al.: Software Product Line Engineering, 2005
[12] AUTOSAR, http://www.autosar.org
[13] ISO 26262, Draft International Standard, 2011