Using architectural constraints to drive software component reuse while adding and enhancing features

In a global software engineering team

Susmita Vaikar, Madan Mohan Jha
Siemens Technology and Services Pvt. Ltd.
Pune, India
susmita.vaikar@siemens.com, madan.jha@siemens.com

Felix Brunner
Siemens Schweiz AG,
Zug, Switzerland
brunner.felix@siemens.com

Abstract—**we develop integrated systems that consist of software and hardware components with a lifespan ranging from 10-15 years. During the life span of these systems, market needs change significantly due to technological advancements, environmental needs, and cultural preferences. Cost of change of software vis-à-vis hardware is a big driver which often leads to the introduction of change to software for meeting evolving market expectations. The biggest advantage of software-'easy adaptability'-is also its biggest drawback, because it makes software susceptible to change. Hence, designing software is extremely challenging specially in Globally Distributed Software Development (GDSD). In this practice paper, we share our approach of leveraging the constraints of software architecture, the challenges encountered and lessons learnt which enabled higher software reuse when adding and enhancing features while reducing overall costs and shrinking time to market for a Globally Distributed Software Development team.**

Keywords—*Software Re-usability*; *Architectural Constraint*; *Architectural Challenges; Tradeoffs; Globally Distributed Software Development;*

I. BACKGROUND

The integrated systems we develop are continually becoming more software intensive, whereas in the past they were more hardware intensive. Furthermore, since several years, our systems have been evolving while growing both in size and complexity.

Software architecture therefore plays a leading role and it has become a central artifact in the life cycle of our systems, because they provide the various stakeholders with an overview of the organization of these systems. Software architecture is defined as "the set of structures of a software system, necessary for reasoning about it…(and) is composed of software entities, the relations between them as well as properties of these entities and relations" [1]. Software architecture makes both the components of a software system and the dependencies between these components explicit [2]. By the year 1990 the term "software architecture" began to attract substantial attention both from the research community and from the industry [3]. The challenges to create evaluate and maintain these huge systems have greatly stimulated the growth of the field of architecture. The importance of software architecture for large and complex software systems can be explained by the following reasons [4].

- Mutual communications: Most systems stakeholders can use software architecture as a basis to understand the system, form consensus, and communicate with each other.
- Early design decisions: Software architecture is the earliest artifact that enables the priorities among competing concerns to be analyzed. Such concerns include tradeoffs between functional and non-functional aspects such as performance, security, maintainability, and modifiability. Additional competing concerns can be cost of current development vs. future maintenance costs or functional completeness or tradeoffs between technical and non-technical aspects such as time to market or budget.
- Transferable abstraction of a system: The model of software architecture is transferable across systems. In particular, it can be applied to other similar systems and promote large scale reuse.

While software systems are evolving, architecture decisions and principles need to be followed. Changing architecture decisions is difficult and it requires a careful impact analysis taking the reasons for past changes into account.. Therefore documenting architecture decisions including the reasoning for the decisions is an important activity in software development processes [5]. Architecture constraints are among the most important descriptions encountered in the documentation of an architecture decision. [7]

An architecture constraint represents the specification of a condition which an architecture description must adhere to in order to satisfy an architectural decision. Therefore architecture constraints play an important role in design decisions and architecture validation. When designing software architecture, the decisions and constraints need to be connected to business goals. Design decisions are often made for non-technical reasons: strategic business concerns, meeting the constraints of cost and schedule, using available personnel, and so forth [8].

2329-6313/16 $31.00 © 2016 IEEE
DOI 10.1109/ICGSE.2016.23

II. INTRODUCTION

Typically the lifespan of our integrated systems ranges from 10-15 years. During its life span a system passes through four different phases (1) Introduction, (2) Growth, (3) Maturity, and (4) Decline.

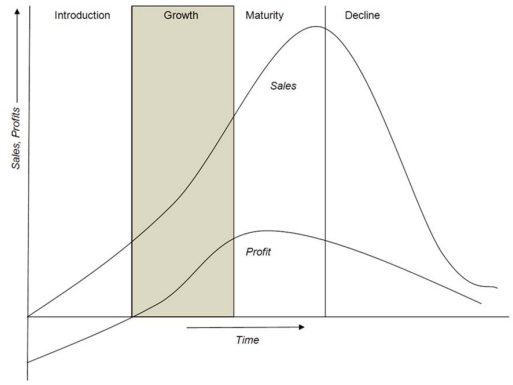

Fig. 1. Product Lifecycle [13]

Market needs for integrated systems are changing significantly due to environmental, cultural, and technological advancements. The cost of change of an integrated system depends on the lifecycle phase it is in. The growth phase for our products is critical and challenging for two reasons: 1) the product has to differentiate itself in the presence of intensified competition that along with market requirements guide which features need to be added to a product, 2) the product undergoes a transition due to large inflow of requirements.

We work on a product line platform which is in growth phase and developed across 3 continents (Europe, Asia, and North America). We provide multiple applications based on this platform and are continuously extending the scope of the product line. Hence it is important to have a structure in place which will act as a guideline for adding new features for all teams across the geographies.

In the subsequent sections we will detail such a structure along with challenges we faced and the lessons we learnt.

III. THE FOUR PILLARS

We designed a four pillar approach for using architectural constraints to drive software component reuse while adding or enhancing features. These four pillars are: (A) requirement engineering analysis based on why vs. what, (B) Variant analysis, an efficient approach for arriving best suitable design option (C) Key architectural decisions and tradeoff analysis for effective reuse, (D) Software component reuse plan.

A. Requirement Engineering:

For our product line requirement engineering is challenging because requirements from different applications are analyzed and platform requirements need to be derived. On the other hand clear and valid requirements are a prerequisite for systematic software design and development since errors at an early stage propagate through the development process and are difficult to resolve later. Requirements elicitation process deals with ambiguity, informality, incompleteness and inconsistency, in which the "knowledge" of the requirements is not clear [9]. Requirements can be classified as functional requirements (FRs) and non-functional requirements (NFRs), e.g. reliability, maintainability, and performance. These requirements are key inputs to software architecture design [10]. We follow standard requirement engineering processes which are (1) requirement elicitation, (2) requirement analysis, (3) requirement specification and (4) requirement validation.

1) Requirement elicitation: We document all the possible inputs gathered from the market/key users and different stakeholders. Because our product line is replacing a set of legacy systems, these stakeholders include key-users and product managers of these systems. The global nature of the product also makes it necessary for requirements to come in from different market segments.

2) Requirement analysis: In the analysis phase we do not look to documented requirements in isolation but rather use multidimensional inspection approach (see fig. 2). We evaluate requirements against legacy and competitor systems, study technological advancements, market needs, feasibility and estimate cost of realization. Based on this initial analysis we decide together with stakeholders which requirements are likely to be realized in the next release and therefore need further evaluation. These candidates go through a segregation process where we study commonality and variability and categorize the requirements based on whether they fall into the scope of the platform or are unique to a single application or product. Requirements with platform impact may be further split into an application specific requirement and a derived technical platform requirement. All platform requirements are ranked according to business goals. Application requirements that are assigned to a separate application team for further analysis. Non-functional requirements are derived using a scenario based approach.

3) Requirement specification: All the requirements which have gone through a multi layered filtering process during

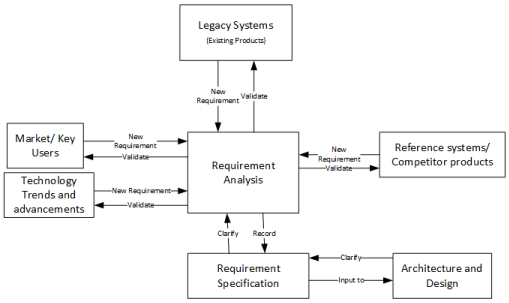

Fig. 2. Multidimensional Inspection

analysis are further refined in cooperation with stakeholders and documented in the scope of the current platform release.

4) Requirement validation: During validation phase we check for requirement attributes such as completeness, consistency, feasibility and testability.

B. Variant analysis:

There can be multiple design options for a solution that fulfills a certain requirement in a product line. Variant analysis is the process of arriving at the most suitable design option by collecting alternatives, documenting and evaluating them. The evaluation can be done conceptually, but often requires prototype development to evaluate feasibility, advantages, and disadvantages of different design alternatives. Clear documentation of the results of the variant analysis has been useful to us in below mentioned cases:

- Multiple valid design alternatives can exist and a decision of these options may be deferred. Here a documentation of the analysis avoids the repetition of design work during later project phases.
- Design options in a product line platform impact design options for applications of the product line. A design option can enforce constraints or it can serve as a blueprint for product design. The architecture for a product line may encompass a collection of different alternatives for dealing with commonalities and variations among products. Capturing these alternatives and the rationale for each alternative enables the team constructing a product to have a list of potential solutions to choose from. [11]

The variants will have different advantages and constraints that need to be evaluated in context of the existing architecture. While doing variant analysis for a new requirement for our product line the selection of a suitable variant is challenging. In the global software development scenario multiple stakeholders are involved for arriving to a decision. The use of the following criteria supports the selection process:

- Quality attributes of the architecture and their rank. We use a set of quality attributes documented in a utility tree. For every quality attribute we have collected a set of representative scenarios that are used while assessing a design alternative.
- Ease of integration with the existing architecture and its components. An architecture that encompasses a large number of different design patterns can easily become inconsistent an incomprehensible over time A design that integrates seamlessly into the existing architecture avoids the inevitable complexity
- The percentage reuse vs. build which influences the time to market and maintainability. Reuse by adapting existing components may increase the complexity of an individual component design but at the same time reduces the complexity of the overall architecture by eliminating duplication associated with multiple designs. On the other hand duplication may be acceptable if the complexity of reuse is too high.
- Upcoming technological and easy adaptation of them. While some technologies provide clear benefits and their use in architecture is advantageous; the indiscriminate inclusion of new technologies, may over time, lead to an excessively complex architecture. We track technologies at organizational level and govern their use by a technology roadmap.
- Impact on applications built on top of the product line. Consistency among design options and the resulting simplicity is a decisive factor in making a product line platform usable for application and product development.

In specific cases variants may satisfy all the points above but introduce some constraints on the requirements. In such a case we may restructure a requirement in cooperation with key stakeholders. Variant analysis requires striking a balance between different options, their functional aspects related to an individual requirement, and non-functional aspects related to the overall architecture.

We had a case for the development of a new feature with two design variants:

- A new feature developed from scratch resulting in the development of an additional component.
- The new feature developed by reuse and adaptation of existing components.

The development from scratch had greater flexibility in terms of accommodating the specifics of the requirement but would have a higher time to market and higher future maintainability costs. The complexity of adapting an existing component was lower than the complexity of managing two independent solutions.

Reuse of existing components had limitations in functionality but advantage of significant savings and quick time to market. At this point we looked back at the requirements to identify the points that were not fulfilled because of the reuse and look back at the why vs. what analysis to identify possible alternatives to achieve the goals which the new requirement intended to realize. This resulted in identification of constraints and new ideas for achieving the goals. The requirements were modified in agreement of key stakeholders.

C. Key architecture decisions and tradeoffs

Quality attributes of large software systems are greatly influenced by the system's architecture. These quality attributes often have impact on each other and cannot be analyzed in isolation. For example, modifiability and flexibility affects performance, security affects usability, and performance and complexity affects maintainability and subsequently cost.

If we simply optimize for a single quality attribute, we may ignore other attributes of importance. Even more significantly, if we do not analyze for multiple attributes, we have no way of understanding the tradeoffs made in the architecture—places where improving one attribute causes another one to be compromised. [12]

The architecture evolves with new requirements that are introduced. Before taking a design decision, we considered following key aspects

- Develop common understanding of the business goals, the requirement, and design decisions among key stakeholders. If the rationale behind design options and decisions is explained to key stakeholders, they might be more willing to compromise on functionality and adjust requirements.
- Constraints, key tradeoff decisions of the existing architecture. Past decisions are required to judge whether a new design decision is consistent or leads to inconsistency and design erosion.
- Compare existing architecture against the key quality attributes of a new requirement. During variant analysis, functional aspects often take precedence over non-functional aspects. This typically leads to unsatisfactory solutions with a penalty of high cost for change at later stages.
- List of architectural sensitivity points. Knowledge about the sensitivity points of the architecture is needed to judge the risks of a design option and is a decisive factor whether early prototyping is required to validate a critical decision.
- Brainstorm on different approaches, rationale and their impact. Stakeholders such as product managers, requirement engineers or architects often believe that there is only "one correct design option". Brainstorming helps opening up the solution space.

We followed the above mentioned steps in one of the case where we had new requirement which imposed strict conditions in terms of latency and the reuse candidate could not satisfy the strict condition without change. We brainstormed to identify and prioritize workflows with acceptable latency limits along with trade-off analysis of quality attribute for each workflow which helped in identifying various solution options. While it was never in doubt that new functionality could be provided and the additional flexibility in the component should not decrease maintainability significantly, prototyping was required to evaluate whether the strict latency conditions could be fulfilled through redesign of an existing component.

D. Software component reuse plan

It is important to use a structured approach when taking reuse decisions. We are facing additional challenges due to the fact that design teams are located at different sites, nevertheless design decisions with global impact need to be coordinated. Following are the highlights of our approach in assessing re-use candidates.

- Component repository - We keep a global repository of software components. This repository is constantly updated and captures key aspects such as responsibilities, non-functional characteristics, assignment to layers and interfaces.
- Architectural and design tradeoffs – Decisions taken as part of variant analysis including tradeoff analysis are captured and documented in the repository.
- Technical debt and limitations – Design limitations and technical debt are documented. On one hand this serves as key input for a team whether a component can be reused, on the other hand the level of reuse can determine the priority to address limitations and technical debt.
- Design and Product Quality Objective - Captures the quality objectives for a component based on its importance, e.g. the expected level of reuse or the impact of failures.

IV. CHALLENGES

We applied the four pillars approach when we had to add significant functionality to our product line platform and faced the following challenges.

The requirements specification was incomplete. The specification was mainly driven by existing solutions in legacy products. Significant quality aspects such as performance and usability where not covered. The commonality and variability between different domains supported by our product line platform where not sufficiently understood.

Product Management that is responsible for the requirements specification had undergone recent personnel change and is located in a different geographical location.

We addressed these deficiencies by inspecting legacy products from all relevant domains. We asked specifically for customer feedback on existing solutions and derived additional quality requirements thereof. We organized regular meetings with product managers to discuss the additional knowledge we gained during the analysis. During these regular meetings we often had to ask for the why vs. what as initial versions of the requirements specification where focused on a single solution.

During the variant analysis we evaluated a component as a reuse candidate because it provided the majority of required functionality and promised significant savings in development and therefore a faster time-to-market. The challenge was to evaluate whether the existing component could be adapted to satisfy higher performance requirements without introducing unmanageable complexity.

We started with the development of prototypes to evaluate whether the requirements could be met and assess the impact of change on the existing code base. During this activity we realized that we could not satisfy all requirements when re-using an existing component. We had to introduce additional constraints to the requirement. In order to convince product management to accept a solution that does not satisfy the original requirement and deviates from the functionality provided in legacy products we had to explicitly state the tradeoffs involved. Full functionality vs. a simpler solution with significantly reduced software maintenance costs and better performance. In addition we could demonstrate that reuse would also lead to a more consistent overall product with better usability.

V. LESSONS LEARNT

- Requirement engineering is an iterative person driven activity. Evolves with further knowledge
- Requirements evaluated from all dimensions and filtered in agreement with key stakeholders add value to the product line.
- An architecture which is open to change and evolution is beneficial for any product. Simple, understandable and consistent architectures are inherently open for change. Flexibility and the resulting complexity must be limited to places where a concrete need for variability exists.
- Every step in the process of architecture design and implementation can and must be allowed to go back to the requirements for reconsideration.
- Documentation of past and current design decisions comes in handy
- Early use feedback proved helpful to uncover issues with having a solution defined with too much legacy system in mind
- Variant analysis is a very important step and provides inputs for future variant analysis and component re-use. Capturing of the results is needed to understand and evaluate historical decisions when evaluating new requirements.
- Architecture tradeoff analysis not only helps in balancing the quality attributes at the time the architecture is created but adds value for addition of new requirements and evaluating the possibilities for component re-use.
- Cautious evaluation of re-use candidate against the quality attribute of the new business requirement helps in creating re-usable prototype and also saves time and cost
- Component re-use has multiple advantages like time to market, consistency, learnability (reduce training efforts) and especially maintainability, but care has to be taken to avoid component re-use in cases of widely diverging functional and more important non-functional requirements.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice, 3rd Edition. Addison- Wesley, 2012.

[2] Mourad Oussalah , Software architecture: Principles, techniques and tools.By Copyright c 2013 John Wiley & Sons, Inc, Chapter 2 SOFTWARE ARCHITECTURE:ARCHITECTURE CONSTRAINTS, p1

[3] R. Kazman. Software Architecture. In Handbook of Software Engineering and Knowledge Engineering, S-K Chang (ed.). World Scientific Publishing, 2001.

[4] P. C. Clements and L. M. Northrop. Software Architecture: An Executive Overview. CMU/SEI-96-TR-003, 1996

[5] Philippe Kruchten, Rafael Capilla, and Juan Carlos Duenas. The decision view's role in soft- ware architecture practice. IEEE Software, 26(2) :36–42, 2009.

[6] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. Documenting Software Architectures, Views and Beyond, Second Edition. Addison-Wesley, 2010.

[7] Chouki Tibermacine, Christophe Dony, Salah Sadou, and Luc Fabresse , Architecture Constraints as Customizable, Reusable and Composable Entities

[8] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, Jeromy Carriere , The Architecture Tradeoff Analysis Method

[9] Jaya Vijayan and G.Raju , A New approach to Requirements Elicitation Using Paper Prototype by

[10] Lin Liao ,From Requirements to Architecture: The State of the Art in Software Architecture Design

[11] Felix Bachmann and Len Bass ,Managing Variability in Software Architectures

[12] Rick Kazman, Mario Barbacci, Mark Klein, S. Jeromy Carrière , Experience with Performing Architecture Tradeoff Analysis

[13] http://www.referenceforbusiness.com/management/Or-Pr/Product-Life-Cycle-and-Industry-Life-Cycle.html