Achieving Sustainable Business for Industrial Software Systems

Pia Stoll, ABB Corporate Research, pia.stoll@se.abb.com,*
Anders Wall, ABB Corporate Research, anders.wall@se.abb.com

The session type selected for the paper presentation:

1) "Dialogue" Session	☐
2) "Author's Presentation" Session	☒

Achieving Sustainable Business for Industrial Software Systems

Pia Stoll, ABB Corporate Research, pia.stoll@se.abb.com,*
Anders Wall, ABB Corporate Research, anders.wall@se.abb.com

Business_process(es) and software_engineering for development of software for complex_systems; impact_of_change on quality_cost; organizational_change.

INTRODUCTION

Sustainable development of industrial software systems with controllable outcome in terms of cost, schedule and quality despite changes originating from new technology, stakeholders' concerns, organization, and business goals during long life-times is a challenge. Unruh [17] has argued that numerous barriers to sustainability arise because today's technological systems were designed and built for permanence and reliability, not change.

Sustainability is a characteristic of a process or state that can be maintained at a certain level indefinitely. The implied preference would be for systems to be productive indefinitely, to be "sustainable." For instance, "sustainable development" would be development of software systems that last indefinitely. Author Michael Pollan [13] has defined an unsustainable system simply as "a practice or process that can't go on indefinitely because it is destroying the very conditions on which it depends.

There are several factors obstructing the sustainability of the software development process:

- Competing concerns from various stakeholders affect the system and the winner among the concerns is not always the most logical. For a mature software system most probably political concerns will compete with functional concerns and affect the system.
- The system's software qualities are exposed to change, e.g. the introduction of faster multi-core processors might solve performance issues outside the scope of the architecture and therefore the focus and mission of the architecture shifts to other issues.
- The business goals of the system are exposed to change. This happens when the management shifts the focus from increase of quality to cost cut and thereby changes one important business goal for the system.
- The technical environment and organization structure change. A new platform or distributed development might be unavoidable and therefore puts requirement on change for the system.

If these factor where possible to control and a stable balance of cost, schedule, and quality outcome of the software system was achieved, the system would be a sustainable software system. The development of the software system would deliver required quality to the customers' satisfaction at the desired scheduled and cost indefinitely. However unrealistic this might seem it is truly the goal of sustainable software development. The cost is a very important measure since a long-lived system can be achieved at a high cost but this would lead to an unsustainable development process which would eventually collapse.

Since software development is considered an art involving people and people communicating a sustainable system model must include influences from people, architecture, hardware, software, communication and unpredictable changes in form of; stakeholders' concerns' changes, technology changes, business goal changes, and organizational changes. With all influences included in one model it would be desirable to be able to predict or at least reason about the outcome of the system; cost, schedule and quality.

The remaining of this paper is organized with a short overview of related work in the section "Related Research" and the issues important for sustainable industrial software systems is given in section "Issues for Sustainable Business". The paper is concluded in the section "Conclusions" followed by a short description of further work in section "Future Work".

RELATED RESEARCH

The importance of technical, business, and social influences on software architecture is discussed in [1] and the relationship among the technical, business, and social environments that subsequently influence future architecture is called the architecture business cycle (ABC). The ABC focuses on the creation of software architecture and the maintenance of the architecture and conformance of the system to the architecture, however, the ABC does not handle sustainable system issues where it's possible that the architecture has to change during the system's lifetime. An attempt to address sustainable systems can be found in [10] where the integration of established engineering methods with a development organization's life cycle is discussed. Here the Attribute Driven Design (ADD) method, [19], and the Cost Benefit Analyze Method (CBAM), [9], are suggested as means for the architect to design and chose appropriate architectural responses to the new challenges during the software development life cycle. The methods are preferably used in the development phase and the Architecture Trade-off Analysis Method (ATAM) used after the system is released and the stakeholders want to discover risks and sensitivity point in the architecture related to business goals.

For the change requests entering the system after its release the stakeholders have to take a decision if they are worth implementing or not. In an article from Boehm [4] it is argued that software engineers should look at proposed changes to software systems as investment possibilities and calculate on the value of investing in those changes with methods similar to the methods in the investment economics, e.g. option theory. Especially the value of the success-critical stakeholders concerns should be considered important. For the sustainable software system this would mean that the software engineers have to be updated on who is a success-critical stakeholder and how to calculate the value of his/hers concern's implementation. The calculation could also serve as guidance to what concerns should be allowed to enter the system as change requests. However calculating a correct development effort for a proposed changer request is very difficult. Joergensen [8] has showed that software project cost estimation uncertainty assessments are frequently based on expert judgment, i.e., unaided, intuition-based processes and not on formal models. His guidelines suggest, among other things, that the most promising strategies are not based on formal models, but on supporting the expert processes.

The implementation of change requests also have to have support in the development process. The process has to support unpredictable change requests as well as support their fast realization. The Scrum [15] development process has gained a lot of supporters as it's a light-weight process with a strong connection to agile development methods. Scrum considers the software development process to be a chaotic empirical process which requires close watching and control, with frequent intervention. A scrum software project is controlled by establishing, maintaining, and monitoring key control parameters. The key control parameters are backlog, issues, risk, problems and changes - task level management is not used. However in [5] it is argued that agile development methods are not well suited to large development organizations such as those evolving sustainable software systems. Scrum identifies the most important stakeholders and these success-critical stakeholder's concerns are implemented at first. This is similar to Ruhe and Saliu [14] who describe the release planning approach based on the features' internal dependencies, the resource constraints and the stakeholders' importance.

In [20] the uncertainty principle of software engineering (UPSE) is stated as "Uncertainty is inherent and inevitable in software development processes and products". The software development is described as a complex human enterprise carried out in problem domains and under circumstance that are often uncertain, vague or otherwise incomplete. The principle of uncertainty is also valid for those changes entering the development organization which are considered unpredictable in time and consequence. The control of the sustainable software development despite the UPSE is what makes the sustainable software development challenging.

ISSUES FOR SUSTAINABLE BUSINESS

The system architecture provides a context for the software architecture and includes, beside software architecture, also hardware and people. System quality attributes and business goals influence the system architecture. The influencing factors which are factors affecting the architecture part of the stakeholder concerns [16] and include trends, technical environment, previous experiences, market demands etc.

The influencing factors change over time and hence the stakeholders' concerns change over time. The influencing factors impact and/or put requirements on system quality attributes and business goals. This leads to that the system quality attributes change as result as well as the business goals. Changing business goals can lead to changing enterprise architecture and changing development organization as business structures and business processes.

Since all these changes come from outside the software system they are uncontrollable and unforeseeable. When building software architecture from start it may be possible to build in support for foreseeable changes but not for an unforeseen change, e.g. a sudden organizational change.

Technology

What makes software especially difficult to develop for sustainable system is that software and hardware themselves are not sustainable. Software technologies, tools, architectures like the World Wide Web, languages like C and C# change the software engineering culture in which system builders operate and learn.

In many cases the demand from the customers on smooth updates preferably in a running plant regardless of what changes occur over time translates into a requirement on backward compatibility. Backward compatibility also concerns hardware, where the customer might run the system on hardware no more available on the market.

For long-lived systems typically the components from which the system is built, have shorter life-cycles than the complete systems. Many components in a large and complex software system are acquired from third-party developers. Consequently, a system provider has no or limited control over the complete system (e.g. no access to source-code). Hence, it is very important to continuously monitoring the sub-suppliers roadmaps and to have a tight and sound relation with them. By doing so, a company have the possibility to react well in time before a particular component or technology for which the development organization has no control over gets obsolete. The fact that software technologies and commercially available software components have shorter life-cycles than what is required for the system is something that needs to be considered when designing the architecture.

Typically the life-cycle of a software product can be divided into three phases: initial design (I), evolution (II), and end-of-life (III) (see Figure 1). During the initial design phase the requirements are usually well-known and the development of new functionality requires relatively little effort. In the evolutionary phase the requirements that were not known in (I) are introduced and the effort for developing and implementing these requirements require higher effort, since consideration must be taken to what already exists in the system. The architecture developed during initial design does to a large extent define what is possible in later phases from an economical point of view.

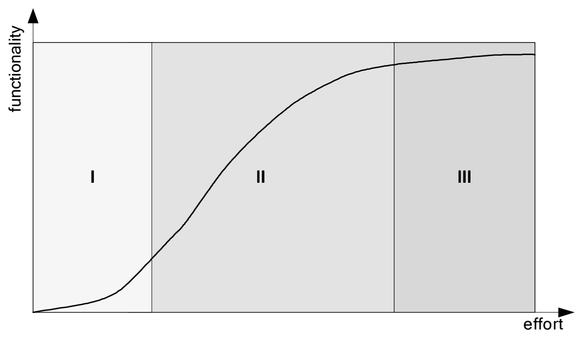

Figure 1 Product life-cycle phases

It is important to find a balance between upfront investments in, e.g. software architectural design, and time-to-market for software development in sustainable complex industrial systems in the perspective of a product's life-cycle. By diagnosing a system's life-cycle phase in terms of trends in crucial organizational measurements we believe that it is possible to quantitatively motivate efforts in improving fundamental software qualities in order to prolong a system's productive life-time. A typical trend in an organizational measurement could be the increasing number of person-hours invested related to the decreasing number of function points delivered. This could be an indication of a system being in the end-of-life phase (III).

Even though technology evolves in a high pace, business specific logic does not. Operating systems and hardware changes all the time but the basic principles for, e.g. control the motion of a robot, evolves slower. Another example is the paper production. The chemical process behind paper production will not change as it's defined by physical parameters and reactions. The control algorithms, which are part of the business logic, involved in controlling the pressure, strain and so on will continuously be refined but not experience major change. Usually there are great investments in the business logic and the investments are secured by intellectual property claims, so it is important to make as much as possible out of these investments. This is where we have the core competence, and the core business. Returning to the core business has proven to be successful for many companies where ABB is one of them. ABB returned its focus to automation and power distribution after some years with a broader scope. Isolating the business logic in a way that enables the technology around it to evolve with the least possible cost is crucial. The statement may seem easy enough but for researchers who have been using FORTRAN for their algorithms because its ability to process a huge amount of control parameters fast and that now have the possibility of using Matlab algorithms translated into C# just as efficiently it's not that easy. Should they now remodel the process in Matlab because in the long run C# offers more advantages than FORTRAN? What's the return of investment, the ROI, value of the change?

Organization

According to [1] there are three classes of organizational influences on software architecture;

- Immediate business: An organization may have an investment in certain assets, such as existing architectures and the products based on them.
- Long-term business: The architecture can form the core of the long-term infrastructure investment to meet the organization's strategic goals.
- Organizational structure: The organizational structure can shape the architecture such that the division of functionality aligns with existing units of expertise.

For sustainable systems there is a challenge in creating a sustainable architecture possible to implement under these three different organizational influences. There will be shifts in organization influence inside a development organization, e.g. if distributed development is introduced. In this case the distributed development could for instance put requirement on the architecture to support isolated module development. Another example is if the architecture suddenly has to support the migration of several products into one, as may be the case when a company acquires another company. For this case the shift in organizational structure goes from immediate business to long-term business. Development organizations often have to deal with drastic shifts like this without the customer noticing any major differences in actual system software quality.

Recognizing that change requests are something normal and that deviations from predictions will occur for a sustainable software system, the question is how to act upon them. Should a change in stakeholders' concerns toward more secure system always respond in that the system is optimized for security? Or will this be in conflict with business goals as e.g. making the system available over internet?

In traditional control theory [12], optimization theories have been developed to optimize the system parameters for stability. Something similar is needed for sustainable software systems in order to make the right system decisions in terms of economics, architecture, technology and people. There are many states that can be controlled and/or observed for a sustainable software system model:

- Software architecture – The design and the infrastructure of the system
- Software technology – The various technologies used as a technical base, such as programming environment, operating system and middleware.
- Software components – The various proprietary and commercial components used to realize the system, examples of components are user interface, user management and transaction managers.
- Hardware – The core of the system where the software is running
- Software communication – everything regarding communication including compatibility with other vendor products, communication hardware, communication stacks and redundancy concepts.
- People interaction – Most industrial systems have people that interact with them and how this is performed is one key to the operation of the whole system.
- Development processes – Processes influence the organization and the architecture and the opposite.

The two last states, people interaction and the development processes, might be the hardest to control since they include human psychology. In [3] programming accidents are examined, i.e., models, methods, artefacts, and tools, to determine that each has a step that programmers find very painful and consequently avoid or postpone. The avoidance or postponement disturbs the processes in a not controllable way and leads at the worst to uncontrollable cost, schedule, and quality outcome.

But before the change request reaches the development stage it has to be approved and there is various way of handling change requirements. In [7] a decision support theory in form of real options theory is suggested for guiding investment decisions regarding a change in the software. Typically the option theory calculations could serve as input to a change request board.

During the lifetime of a long-lived system there will be a turn-over of engineers. The engineers possess competence and know-how concerning the system. Typical examples of crucial know-how is the intention and rational behind certain architectural decisions. As engineers come and go through the organization there is a great risk that this knowledge is lost. As a consequence, poor design decision may be taken during a system's evolution which contributes to shorten the productive phase of the sustainable systems. A proper architectural documentation is one way to minimize the risk of competence drain due to turn-over of engineers. Yet again the human psychology aspect enters the field since software developers often find documentation a very painful step and avoid this as far as possible. When documenting software the people doing the documentation has to find it meaningful and ultimately, such documentation has to have some notion of intention, i.e. rationales for architectural decisions [21].

Market

It's not only customers' expectations that change over time. Also a company's business goals change, e.g. penetration of new markets. Every company has its own set of business goals and to achieve a common perception of the goals, it would be beneficiary to generalize them. One approach is presented by Bass and Kazmann where they have categorized the business goals from a number of ATAM evaluations [2]. Their five categories are; 1) "Reduce total cost of ownership", (2) "Improve capability/quality of system", (3) "Improve market position", (4) "Support improved business processes", and (5) "Improve confidence in and perception of the system".

Typically there will be a movement between quality focused business goals as (1), (2), and (3) and functionality focused business goals as (3) and (5). A "fresh" software system is typically more focused on "Improve market position" and "Improve confidence in and perception of the system". New functionality is then released to customers and feedback from the release in form of change requirements and trackers leads to yet more new functionality. When the software system has grown to a certain extent the focus might shift to quality focused goals as "Reduce total cost of ownership", and "Improve capability/quality of system".

The challenge lays in balancing the shift in business goals with their interpretation to software quality goals and functionality requirements. For example "Reduce total cost of ownership" can mean outsourcing parts of the development and this puts high requirements on the modifiability and testability quality and also on software development processes different to in-house development [11].

Another example is the conflict of the shift towards "Reduce total cost of ownership" including the tactics to use standard hardware. If the market differentiators for the product are high robustness and backward compatibility, it means the robustness issue has to be solved with standard hardware and the backward compatibility issue with non complex architecture in order not to implement expensive development. This is truly a challenge. The customer's perception of the system should be the same, only with updated software and hardware. Industrial systems have customers running legacy hardware which have no intention or motivation to shift hardware to the latest technology. For system developers the customer's hardware puts requirement on the software to be backward compatible with the legacy hardware as well as backward compatible with legacy software.

It is not uncommon for industrial software system to have a few dominating customers who demand certain system qualities. In this case the challenge lies in to what extent the system producer can tailor the system to please one dominant customer before the other customers object to not getting their requirements met or having to pay for qualities they don't require. We have seen examples where a few dominant customers have driven a system to be too costly compared to competitors offers. The reason is that the system provides a lot of functionality which are not specifically requested by the majority of customer categories, but requires more expensive hardware infrastructure which contributes to the cost. However there is also an advantage with a large dominant customer. They provide the means for the rework of one system to an extent not possible otherwise, which in the CelsiusTech case proved very successful. In the case of CelsiusTech [6], the unpredictable change in the form of the simultaneous awarding of two massive contracts (each of which was for a system beyond anything the company had ever attempted) led to a complete redesign of the system architecture based on the core assets. The new product-line architecture was the entry to new business areas not previously accessible.

CONCLUSIONS

This paper has described the challenges for the development of sustainable industrial software systems. The most important factor to recognize is the factor of time and its effect on system development since industrial software systems often have long lifetimes. The second factor to recognize is that change in organization, technology, and market over time is something inevitable and that the development has to calculate for this. The third factor to recognize is that changes are not always predictable or foreseeable and that a static system could have difficulties to host unpredictable and unforeseeable changes. The forth factor to recognize for industrial systems is that their customers most often don’t want to experience any change since a change requiring knowledge update or process interruptions is costly. The last factor to recognize is that the producer can achieve the desired quality and cost despite unpredictable changes at an unreasonable cost, but this would lead to an unsustainable development process which would eventually collapse.

This leads us to the conclusion that the sustainable industrial software system has to control the cost, quality, and schedule outcome of the system despite unpredictable and predictable changes in organization, market, and technology affecting the system over time.

FUTURE WORK

Future work will include an attempt to establish a sustainable software system model, including measures for the key states important for the control of the outcome of a sustainable industrial software system. In this work software economics will be a key essence influencing the software engineering theory for the model.

References

[1] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley (1998)

[2] Bass, L., Kazman, R.: Categorizing Business Goals for Software Architectures. In: SEI (ed.), Pittsburgh (2005)

[3] Berry, D.M.: The Inevitable Pain of Software Development: Why There Is No Silver Bullet. LNCS 2941. Springer Verlag (2004)

[4] Boehm, B.W., Sullivan, K.J.: Software economics: a roadmap. Proceedings of the Conference on The Future of Software Engineering. ACM, Limerick, Ireland (2000)

[5] Boehm, B.W.: A view of 20th and 21st century software engineering. Proceeding of the 28th international conference on Software engineering. ACM, Shanghai, China (2006)

[6] Clements, P., Northrop, L., Software Product Lines: Practices and Patterns, Addison-Wesley (2002)

[7] Erdogmus, H.: Valuation of Complex Options in Software Development. ICSE'99 Workshop on Economics Driven Software Engineering Research (EDSER1). ACM/IEEE, Los Angeles (1999)

[8] Joergensen, M.: Evidence-bases guidelines for assessment of software development cost uncertainty. IEEE transactions on software engineering 31 (2005)

[9] Kazman, R., Jai, A., Klein, M.: Quantifying the costs and benefits of architectural decisions. In: Jai, A. (ed.): Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on (2001) 297-306

[10] Kazman, R., Nord, R.L., Klein, M.: A life-cycle view of architecture analysis and design methods. In: SEI (ed.), Pittsburg (2003)

[11] Larsson, S., Wall, A., Wallin, P.: Assessing the Influence on Processes when Evolving the Software Architecture. Workshop, IWPSE'07, Dubrovnik, Croatia (2007)

[12] Ljung, L.: System Identification - Theory For the User. Prentice Hall, Upper Saddle River, N.Y. (1999)

[13] Pollan, P, Our Decrepit Food Factories, New York Times, 2007.

[14] Ruhe, G., Saliu, M.O.: The art and science of software release planning. Software, IEEE 22 (2005) 47-53

[15] Schwaber, K.: SCRUM Development Process. OOPSLA 95 Business Object Design and Implementation workshop (1995)

[16] Stoll, P., Wall, A., Norström, C.: Guiding Architectural Decisions with the Influencing Factors Method. WICSA. IEEE, Vancouver (2008)

[17] Unruh , G.C., Escaping carbon lock-in, *Energy Policy*, vol. 30, no. 4, 2002, pp. 317-325.

[18] Unruh, G.C., Understanding carbon lock-in, Energy Policy, vol. 28, no. 12, 2000, p. 817-830.

[19] Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R.L., Wood, B.: Attribute-Driven Design (ADD), Version 2.0. CMU/SEI, Pittsburg (2006)

[20] Ziv, H., Richardson, D.J.: The Uncertainty Principle in Software Engineering. 19th International Conference on Software Engineering (ICSE'97). ACM (1997)

[21] Leveson N. G, Intent Specifications: An Approach to Building, Human-Centered Specifications, IEEE Transections on Software Engineering, vol. 26, no. 1, 2000