

# Software Engineering Research Suffers from the Beehive Syndrome

Mira Kajko-Mattsson  
KTH School of Information and Communication Technology  
Sweden Stockholm  
mira@dsv.su.se

## 1. INTRODUCTION

Many of us, software academics, are like a swarm of bees. We fly where the honey is. The honey is either a research area imposed by a sponsor providing us with research grants or some hype that enables us to come up to the surface as famous researchers.

There is nothing wrong with honey as long as researchers feel properly nourished with fame and all the necessary corners of software engineering research are evenly explored. Fame and honey is the researchers' survival mechanism in this highly competitive research environment. If properly nourished, it may result in high quality research results. If malnourished, by for instance the sponsors determining the research topics, it may substantially delimit the research scope and thereby damage the long-term software engineering research. Unfortunately, the sponsors are the ones who determine the flying directions of the researchers.

Research within software engineering should be beehive-centered, where beehive is the discipline of software engineering and the bees are the researchers and practitioners. To properly support the beehive, the bees should organize themselves and make sure that all the needs within the beehive get properly fulfilled. Unfortunately, some of the bees, not all, suffer from the beehive syndrome. Being injected with various fads and trends, they may demonstrate impaired hearing and vision towards industrial and academic needs. Hence, they do not always find their way back to the beehive.

Software engineering research has not been evenly distributed. It has been mainly focused on the initial software lifecycle phase, the honey phase or the software development phase. Regarding other lifecycle phases such as software evolution, maintenance and retirement, they have been merely treated like a black box. As a result, the software community lacks understanding of the overall software lifecycle and the reciprocal interplay amongst its component phases.

Today, we do not possess a solid basis for defining software lifecycle. Our definitions are unclear and our classifications of software lifecycle and its categories are vague. We have difficulties in identifying the scope of software development, evolution and maintenance, and in drawing a dividing line between them. We cannot even agree on what to call various engineering domains. We have problems when meeting new hypes and when adapting to new technologies. All this is because we have not put enough effort into exploring the whole software engineering domain. We have mainly concentrated on researching on limited parts of it, the honey parts.

## 2. LIFECYCLE

A software system undergoes many different phases from the time when it is conceived till it gets retired. These include development, evolution and maintenance and finally retirement.

Unfortunately, the software community mainly focuses on only development and development processes. It has not put much effort into exploring other lifecycle processes. These do not only include primary processes such as evolution, maintenance and retirement, but also all kinds of support and management processes. Many of them are as good as non-existent within the software engineering research today.

To make research within software engineering evenly distributed, the software community should create a process roadmap. Such a roadmap should list the processes to be managed during the lifecycle, place them on the organizational levels, place them on the software lifecycle phases, identify their role within each organizational level and lifecycle phase, weigh their business criticality and rate their research urgency.

Among the most urgent processes to explore, we include software evolution, scheduled maintenance, emergency maintenance, crisis management, predelivery maintenance, preventive maintenance, software handover (transition), new trends in software development, SLA management, testing, front-end support (1<sup>st</sup> line and 2<sup>nd</sup> line support), release management, risk management, measurement, and documentation.

It is not enough to only identify the processes. As mentioned above, the roadmap should also outline the organizational levels relevant for the lifecycle processes. It is only in this way, we may get a better understanding of the processes and their role within software organizations. The organizational levels do not only include the strategic levels that are mainly relevant in the context of a software development or evolution cycle. They should also consider other levels such as support line levels within support and corrective maintenance, operational levels in the context of emergency maintenance, and, echelons in the context of supporting integrated software and hardware systems [3].

Substantial process knowledge is found within the industry today. Hence, a lot of know-how can be explored from there. This may be however a very challenging task. Many of the organizations possess this know-how in their own way. Not having access to any universal process guidelines, they have been forced to capture their own experiences with their own sweat and tears and put them into their own process models. Hence, their processes, their naming, scope may vary strongly depending on the organizational context, size and complexity. However, they are all based on more or less similar experiences. Therefore, exploring them may provide useful feedback for creating such a roadmap and for exploring lifecycle processes and their practices.

## 3. DEFINITION AND NAMING PROBLEMS

To define lifecycle seems to be easier said than done. This is because there reigns a big confusion within the software community on how to name and define most of its constituents. Many organizations, institutions and authors may have different opinions on their definitions and naming.

Let us take an example of evolution and maintenance. Right now, we have a silent war in which the software community is

split into those who accept the term software maintenance, those who do not and those who are unaware of the problem and hence, they accept whatever the definition is available to them. Those who do not accept the term software maintenance, they do not even mention the word maintenance in their research. They use the term evolution instead. In their opinion, the term “software maintenance” is a misnomer since it states that software should be recovered to its initial correct state. However, after getting updated, the software never steps into its original state [4].

Those, who do accept the term maintenance, on the other hand, either use the term maintenance to refer to all postdelivery evolution and maintenance activities or they use a combined term of software evolution and maintenance [1].

To add zest to the naming problem, the industry uses their own nomenclature which definitely does not agree with the nomenclature as used by the standards and the academia. Most of the industry uses the definitions of development and maintenance as used within other engineering disciplines. They mainly use the term development in the context of new development and continuous enhancement. They use the term maintenance only in the context of attending to corrective maintenance tasks.

Irrespective of how we call what, we have difficulties to define their constituent activity classes and to draw a dividing line between them. In the early software engineering era, we could clearly identify the development phase. Right now, in the era of reuse, code generation, component-based development, model driven architectures, and agile methods, we experience that the distinction between development and maintenance gets blurred.

Regarding the postdelivery activity classes, standards say one thing, researchers another, and the industry goes its own way. For example, standards identify five types of maintenance categories such as corrective, perfective, adaptive, preventive and predelivery. Some researchers identify enhance, corrective, reductive, adaptive, performance, preventive, groomative, update, reformative, evaluative, consultive, and training. Finally, the industry mainly distinguishes between development and maintenance. Their determination of what is software development or maintenance is not always based on any clear-cut definitions. It may depend on business and political reasons as well [1].

The above-mentioned definition problems and lack of a process roadmap lead to the fact that we cannot always trust many of the research results. Many of them are deficient in defining their terminology, in identifying the scope of their research or even in clarifying the terminology they use. Hence, the interpretation of their research results lies in the eyes of the beholder and misinterpretations proliferate. What does it mean that maintenance costs 70% of the total lifecycle cost when the authors have not specified how they understand maintenance and they have not considered support line levels on which corrective maintenance takes place [2].

To remedy the problem, we need a universal glossary that we may refer to when identifying the lifecycle scope. This glossary should not be restrictive. It should still allow researchers and practitioners to use their own terminology. However, it should require that they should correlate their vocabulary to the universal glossary, when need arises. Although we already have a universal glossary today, it seems we have stagnated to further evolve and maintain it.

#### **4. WELCOME NEW HYPES**

Software engineering is continuously bombarded with various hypes. Hypes are built around new overly optimistic methodological or technological innovations flavored with high expectations.

With time, some of the hypes may mature and prove to provide yet another long-term benefits. Therefore, we may accept them as important constituents of the software engineering domain. Some other hypes, on the other hand, may never mature. Hence, they may quickly fade away. Irrespective of their outcome, however, we should always embrace them. We might find their ideas useful and constructive.

Hypes challenge our traditionally accepted norms of developing and maintaining software. If they do not make their voices heard enough, they may vanish unnoticed. However, if they raise their voices loud enough, we will have to listen to them, react and defend ourselves. The fact that we have to defend ourselves may make us realize that there may be other ways of developing, evolving and maintaining software. Some of them may prove useful and beneficial. Hence, we may accept them as new practices. Some of them may prove detrimental. Then, we may reject them and never bother about them again.

The degree of the benefits gained from new hypes varies. At its worst, the software community may be forced to explore its ideas soon to realize that they have nothing to offer. At its best, new hypes may give us a totally new slant and make us realize that we need to change our mindsets with respect to how to manage software systems during their lifecycle.

Clear evidence may be found in the context of agile methods. They challenged many of the aspects of traditional development. Some of these aspects were strongly neglected within the industry, such as for instance, developers’ tests. With the help of agile methods, developers’ tests were placed on the pedestal. Some other aspects were very complex activities, such as, for instance, acceptance testing. With the iterative mindset, acceptance testing has become strongly simplified and greatly improved in some contexts. Finally, the agile way of understanding and changing requirements has made us realize that there may be other ways of embracing change.

#### **5. FINAL REMARKS**

Despite its age of almost half a century, software engineering is not yet mature enough. It still suffers from enormous gaps in the knowledge and understanding of software lifecycle processes and their reciprocal interplay and impact on each other. The gaps will stay if we do not put enough effort into decreasing and/or removing them.

One way of doing it is to create process roadmaps (1) mapping out organizational levels, lifecycle processes and their practices, (2) placing the processes on the organizational levels and (3) identifying their roles within each organizational level and lifecycle phase. The processes should be weighed according to their business criticality and they should be rated by their research urgency. It is only in this way, we may avoid the beehive syndrome of the software engineering research, be certain that we have covered the whole software engineering scope and that no processes are left unexplored. It is also in this way we will be able to create a solid theoretical basis for defining the lifecycle processes and their best practices.

To remedy the problems of definitions and terminology may be a challenging task. However, it may be realizable in cases when the software community commonly makes the effort of organizing itself, creating a common however evolvable nomenclature and making sure that it gets updated as soon as new terminology emerges and the old one fades away.

Definitely, we should welcome hypes. They make us wake up from our deep-rooted mindsets and make us judge them from new angles and perspectives. This, in turn, helps us identify new directions of software engineering or confirm the usefulness of the current ones.

Finally, even if we strive towards defining sound and widely accepted theoretical basis, we should not reject suggestions for changes knocking on our doors. We should embrace them and treat them as the opportunity for confirming our existing theories or revising them or completely changing them due to our increased understanding or changed needs.

## 6. REFERENCES

- [1] Chapin, N., Hale, J. E., Khan, K. Md., Ramil, J. F., and Tan, W.-G., "Types of Software Evolution and Software Maintenance," *Journal of Software Maintenance and Evolution*, 12(1), 3–30 (January-February 2001).
- [2] Kajko-Mattsson., Maturity Status within Front-End Support Organisations, International Conference on Software Engineering, IEEE, Computer Society Press: Los Alamitos, CA, ISBN: 0-7695-2828-7, pp. 652-663, 2007.
- [3] Karim, R., Kajko-Mattsson, M., Söderholm, P., Candell, O., Tyrberg, T., Ohlund, H., Johansson, J., Positioning Embedded Software Maintenance within Industrial Maintenance, IEEE International Conference on Software Maintenance, IEEE Computer Society Press: Los Alamitos, CA, ISBN: 978-1-4244-2613-3, 2008.
- [4] Lehman, M.M., On Understanding Laws, Evolution, and Conservation in the Large Program Lifecycle, *Systems and Software*, 1(3), (1980), 213-221.