Software Climate Change and its Disruptive Weather: A Potential Shift from "Software Engineering" to Vibrant/Dynamic Softology

Imran Ghani[1] and Seung Ryul Jeong[2]
[1]School of Information Technology, Monash University Malaysia
[e-mail: imransaieen@gmail.com]
[2] Graduate School of Business Information Technology, Kookmin University
Seoul, South Korea
[e-mail: srjeong@kookmin.ac.kr]
*Corresponding author: Seung Ryul Jeong

Received May 9, 2016; revised July 13, 2016; accepted July 21, 2016; published August 31, 2016

Abstract

Like natural climate change on the planet earth, the climate in software development environments is also changing (fast). Like the natural weather, the software environment is also disruptive. As the climate experts alert and suggest taking necessary measures to overcome certain challenges to make this earth a safer and comfortable living place, likewise this article also alerts the relevant stakeholders of software craftsmanship about the dynamic challenges that traditional Software Engineering (SE) with purely "Engineering mind-set" is not capable to respond. Hence, some new thoughts to overcome such challenges are shared. Fundamentally, based on the historical evidences, this article presents the authors' observation about continuous shift from traditional "Engineering-based" software development approaches to disruptive approaches – "Vibrant Softology". The authors see the cause of this shift as disruptive transformational force, which is so powerful that it is uncontrollably diminishing the "Engineering-based" approach from software development environments. The authors align it with climate change analogy. Based on this analogy, the authors feel the need to theoretically re-coin the notion of SE to some new term; perhaps Vibrant/Dynamic Softology (VS or DS). Hence, the authors suggest "a new (disruptive and dynamic) way of thinking is required to develop software". It is worth mentioning that the purpose of article and this new theory is not to disparage the notion of software engineering altogether, rather the aim is to highlight the importance of transformation from SE to its next level (perhaps VS/DS) due to the emerging needs in the software craftsmanship environment.

Keywords: Softology, Software Climate Shift, Organizational Weather, Software Engineering

http://dx.doi.org/10.3837/tiis.2016.08.027 **ISSN : 1976-7277**

1. Introduction

1.1. Did "Software Engineering" Truly Exit?

Since 1968, when the term "Software Engineering" was first introduced, software academics and practitioners have been inconveniently using the notion of "Software Engineering (SE)". The term was accepted with a hope that sooner or later "Engineering-based" approaches would be introduced to develop software. However, hesitation had been still there whether to use this term with confidence or not. This hesitation had been so long and so obvious that it took IEEE Standard Committee 22 years to come up with a formal definition of "Software Engineering" in 1990.

Yet, a confident use of the term could not be established. This hesitation has a good reason. When one looks at the non-software engineering domains then one finds that the process of Software creation does not include many "engineering" concepts and practices. The concepts that are part of "engineering" education, for instance precise coverage of discrete mathematics, probability, statistics, trigonometry, geometry and so on are part of only a few standard curriculum of "SE" education (ABET - USA, Criteria for Accrediting Engineering Programs, 2014 - 2015), while most of the curriculum do not teach/use/practice them. So, when we call that software creation is a "Software Engineering", we cannot find the mapping between "SE" and other non-software engineering disciplines. Neither a software designer, developer, tester, team leader or project manager feels like "engineering" while working in their offices. Hence both these concepts do not truly map with each other.

Some might disagree and suggest that "It is the manner of that mapping that matters. We may succeed mathematics, trigonometry and geometry with other domains relevant to our software domain." But, what are those other domains based on theories and laws that exist in software domain? We do not have any theory or law in SE (as yet). This reality was also been noticed by the [1] and [12].

Let us analyze a standard IEEE definition of "SE" and its relevance in today's environment.

***Definition of Software Engineering by** (IEEE Std 610.12-1990):*
"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software"

Based on our observation and also working experience of authors in software houses in several different countries like in South Korea, Malaysia, UAE, Pakistan and UK, it has been seen that, these days, Software is not developed, operated and maintained in a systematic and disciplined manner. Yes, however, quantification element is still valid. In an article, [2] shares the similar views

"The effort of systematizing and/or fully standardizing the process is useless. The rules can be followed, ignored or broken. It's Dichotomy Principle. Minimum systematizing and standardizing and maximum autonomy depends upon the situation team is facing."

We can observe that "Engineering", in its true sense (a systematic, disciplined, quantifiable approach), seldom happens at individual level or at the team level while creating software. However, it does not mean that the things are done in a wrong way by a software company.

Then we will have wrong software product. Though, creating Software needs skills but not so critical skills like what "Engineering" needs. We agree that the scale and criticality of the Software project does matter. For instance, a number of high school students are creating excellent Software without any knowledge of "Engineering" and without following a systematic, disciplined, quantifiable approach. However, the scale of project is small and that is not critical too, such as aircraft, space shuttle or medical systems. This point is arguable.

By nature, creating Software is unique to other systems. The process is not systematic but variant; environment is not disciplined but disruptive. The Software product itself is also dynamic. In order to be systematic the teams are able to plan a lot but are not able to follow all these plans exactly. Plan-driven approaches are not helping now due to frequent requirement change. Different channels (business, end users, Software teams) work on the Software simultaneously, concurrently and randomly.

This problem is further amplified by a major shift in the types of Software applications that businesses require to deliver, from systems of record (big data, cloud, services) to systems of engagement (mobile apps and social media). Such applications must be easy to use and high performing. Because systems of engagement are used directly by customers, they require intense focus on user experience, speed of delivery, and agility — in other words, a vibrant approach is needed.

In such scenarios a systematic and disciplined approach is a hurdle than a help.

In order to be sure about this phenomenon, let's try to identify the significant correlations between the two environments: Software development environment and civil engineering environment. Based on **Table 1**, it is hard to find the commonalities between the two.

Table 1. Uncommon distinguishable properties of the two environments

#	**Software Creation Environment**	**Civil Engineering Environment**
1	Unsystematic, Chaotic, Arbitrary, Disorderly	Systematic, Methodical, Orderly
2	Disruptive, Non-predictive	Disciplined, Well-ordered, Predictive (due to low last minute requirement change disruption)
3	Adaptive-planning (due to high disruption)	Plan-Driven,
4	Flexible Twisting of Process	Non-flexible Twisting, Rigid Process, Smooth
5	Unstable Requirements	Stable Requirements
6	Tidy Environment	Untidy Environment
7	After products, Software Developers are easily accessible to end users	After products, engineers are not accessible to end users
8	Early Feedback Cycle	Late Feedback
9	Easy fixes to broken tools	Hard fixes to broken tools
10	Medium & Continuous Maintenance	Medium & Continuous Maintenance

Table 1 presents the fact that the two compared environments are very different. These differences have good reasons. As mentioned above, Engineering education programs, unlike Software programs, typically require multiple semesters of higher-level mathematics, geometry, trigonometry, calculus and calculus-based theoretical science courses. Engineering programs typically focus on algebra, trigonometry, applied calculus, and other courses that are more practical than theoretical processes in nature. These educational programs create "Engineering Mind-set", which we do not create in "SE" education programs – neither do we use them in real software development.

However, still, some of us emphasize and utilize "partial" Engineering Mind-set (as we are not equipped with such skills and we actually do not need them – please refer to the above example of high school students who can develop working software with minimum skills these days). "Engineering Mind-set" is going to cause more problems because "SE" has become too huge to cover as highlighted by SEMAT [3] and [4]. Let's have another look on a few other uncommon aspects. **Table 2** presents the properties of Civil Engineering products (such as bridge) and Software products.

Table 2. Uncommon distinguishable properties of the two types of products

#	**Software Product Properties**	**Civil Engineering (Non-software) Product Properties (e.g., Bridge)**
1	Highly Re-usable (Function & Data)	Not Highly Re-usable (Material)
2	Cheap Back Up	No Back Up
3	High Industry-based Increment	No Industry-based Increment
4	Less Intense Design	High Intense/Precise Design
5	Mostly Downloadable	Highly Physically Shipping
6	Not Depreciative	Depreciative
7	Easy, cheap and less frequent maintenance	Difficult, costly and frequent more maintenance
8	High Customer Engagement (Fast feedback)	Low Customer Engagement (Slow feedback)
9	Adaptive	Not Adaptive / Fixed
10	Cheap Energy Usage	Expensive Energy Usage
11	Protected	Protected
12	Unlimited Install / Uninstall / Reinstall (in principle)	No Reinstall (in principle)
13	Open source or free software	Nothing is open source or free
14	Global Use (Possible)	Local Use

Based on **Table 1** and **2**, there is no significant correlation between the two environments in which software is developed and also no significant correlation between the properties of non-software products and software products.

The reasons behind these differences, among others, are that there is a big dissimilarity of mind-sets of applying these approaches. In the software development, high customer

interactions (emails, phone calls, walk-in, progress status sharing), and requirements disruptions make the soft development process vibrant (variant), such as a high-velocity scenario to produce software fast, evolving number of end-users (even though product is not 100% complete), as well as competitors. The introduction of new information technology tools, the Internet and the globalization of the markets are contributing to this phenomenon, and nothing seems to reverse the process [5]. While in non-software environment, most of such cases do not even exit where assumption of a controlled environment is too restrictive. So, the rules of *systematic and discipline* proved effective for "Engineering" discipline not necessarily result useful in the Software discipline.

Perrow [6], introduced a two dimensional classification of the technology (**Fig. 1**). The first dimension is the analyzability of the problem varying from well-defined to ill-defined. The second dimension is the task variability, which means the number of expected exceptions in the tasks. According to [5], who interprets this model in **Fig. 1**, Software engineering unlike other forms of engineering, presents decision problems on more than one quadrant. Fuzzy problem definition and high number of exceptions characterize requirements, analysis and design phases. According to [5], only code generation and testing phases are in the engineering quadrant. We, the authors, would like to add that Software Design phase is also in the engineering quadrant, as design is the most important activity in all other non-software engineering disciplines. The main volume of activities of Software resides in the non-routine quadrant.

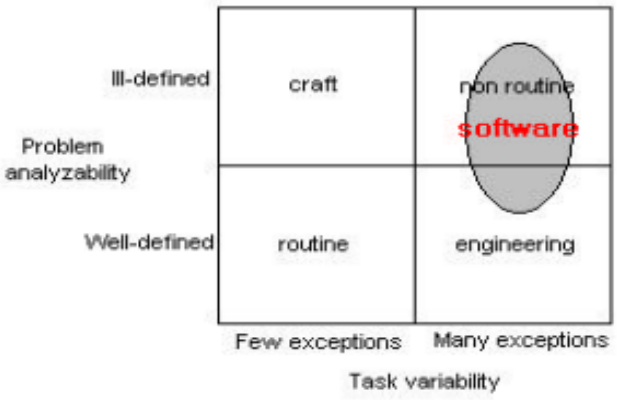

Fig. 1. Classification of Technology

According to [5] software engineering is not the only discipline in this quadrant. We, the authors agree to this point.

This is true that, earlier, many concepts and technical practices were considered as part of "SE". However, they are disappearing from SE now. The disciplines are getting specialized recognition by themselves (at least in theory); it means they were not fit for "SE" discipline. For instance, testing was considered as part of "SE". But, now testing is considered a specialized discipline in itself. Though, it is interesting to notice that many small & medium software companies do not have testers or testing teams. In such companies, developers do the

software testing. However, that is another debate whether we really need testing teams for non-critical systems. The programming is considered a specialized discipline under computer science. None of these necessarily follow the "Engineering mind-set". Instead, new and better technical practices, such as simple requirements, simple design, early customer feedback, fast collaboration, cross-functional teams, are appearing which do not mirror the "engineering" principles.

1.2. Yes, "Software Engineering" Did Exist, but Partially!!

Despite the unclear name, the discipline of "SE" brought with it some "Engineering" concepts partially and for a limited time (though this happens with other fields too, such as cooking). For instance, project planning phase and drawings (architecture and design phase) are the most significant ones in "SE". Let's analyze these two practices of "SE" and try to align with "Engineering".

Firstly, "Engineering" arises in Software application where plans are planned and legislative standards are to be followed and that precisely emerges for a small number of (traditional) systems. Because these applications do not need to change often, organizations can satisfy their customers and their own business needs by delivering only one or two large new releases a year. Such organizations can afford some systematic practices of "Engineering" with Detailed Requirements, Detailed Architecture, Detailed Design, Formal Methods, Detailed Testing and so on. Interestingly, during the SDLC, even such organizations sometimes do not strictly follow the established concepts, processes or methods to develop complex systems. They develop their own approaches that they feel comfortable [7]. For instance, in principle, different types of UML diagrams (sequence diagram, class diagram, state diagram, and so on) should be created separately. However, if you see a discussion among Software Designers on a white board, they mix up different types of UML diagrams. They do not create UML diagrams precisely too. They break the UML "rule" but this adaptive modeling approach helps them understand the system's feature more efficiently.

Secondly, "Engineering" arises in critical Software application where detailed design is inevitable. However, unlike the constructional design engineers who ensure that building actually can stand up, the software engineers are not assertive enough about the software design, because back in their mind is iterative thought. They know if something goes wrong with the software they will iterate to improve it. The luxury of iteration is not available in non-software domains.

Yet, whenever we imagine "SE" the first thing comes into mind is Design of Software. We think this Design is the major magnetic concept that is keeping "Engineering" term associated with Software development. Because, Design is the key in other non-software engineering domains. For this reason, Design provides the best reference point for "SE" and an inclusive, systematic approach for integrating "Engineering" and "Software Engineering" approaches. Let's analyze this Design dominance theory in the next section.

2. Design Dominance in Engineering and "SE"?

The [8] described the history of a number of technological fields as they mature. In his study, he identified four stages through which engineering technologies typically pass on their way to maturity. In each stage, Design is key focus of "Engineering".

- The first stage, in this case, involved adapting the basic concept of the ancient rope suspension bridges into sound *"engineering designs"* with modern materials.
- The second stage involved cautious *"design improvements"* based on experience gained during the first generation.
- In the third stage, engineers pushed the limits of the "existing technologies" until major failures occurred. The civil engineering community studied and analyzed the causes for such failures and developed solutions that then became *"part of the design methodology"*.
- The fourth, mature stage is solidly based on the development and adoption of the lessons learned from past failures and successes. It is obvious that he is referring to the past Design failures.

The question is do we focus so much on Design of Software now a days? In order to find an answer to this question, we have conducted a survey in January 2014 using Linkedin (where participants were from many different countries) and also in a face-to-face meeting with software developers, team leads and project managers from 15 software companies in Malaysia. The respondents had 5-30 years of experience ranging from waterfall to agile development (**Fig. 2**).

The question was simple: Do you use UML Diagrams these days?

Answer Selection:

- No
- Yes
- Rarely
- Often

The 60% Malaysia respondents who had operational/established new practices (such as agile) for 1 year to 3 years or more, replied they do not use UML Diagrams at all.

Reason # 1: They used to draw UML Diagrams in the beginning, but they were not helpful in frequent requirements change.

Reason # 2: They did not have good UML Designers (not sure, once they have him/her, then they fall into the reason # 1 above).

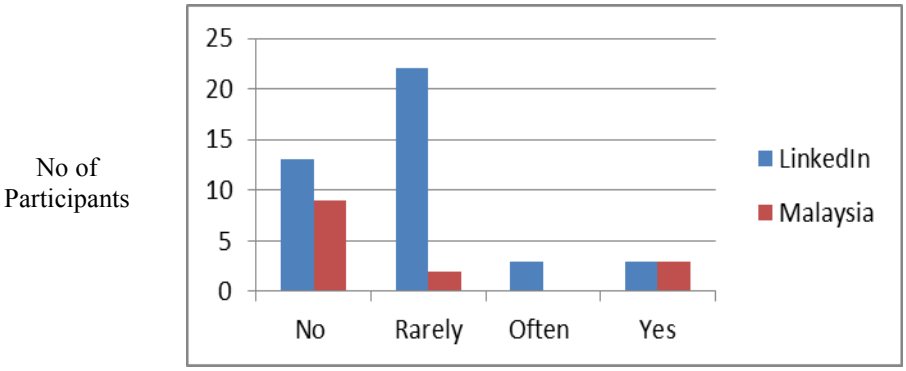

Fig. 2. Survey Response on Usage of UML (January 2014)

The answer received from Linkedin respondents was not much different either – except more respondents said that they rarely use UML diagrams. Majority of current Software teams do not draw heavy diagrams due to - You Aren't Gonna Need It (YAGNI) principle [9]. They relate Software modeling to the way in which Software team perceives and visualizes the problem for understanding.

In an article, "Is Design Dead?" written by [10], author analyzes the usage of Design in Software. Fowler said about OOP that *"Don't draw every class, - only the important one"*. This statement is further explained by him *"The danger of intermingling design with programming is that programming can happen without design - this is the situation where Evolutionary Design diverges and fails."*

The above statement is self-explanatory about software that reflects decision making in software development is easy. While in civil engineering, we cannot think of building a bridge without design. In software development, design is normally used just as a brain storming tool whereas in non-software domains it is part of the product.

In response to our Linkedin survey, as mentioned above, some of the respondents with more than 30 years of experience in Software industry commented as follows

*"Learning how to model, and getting good at it, is a great help in learning ways to think about the complexities of systems. But once learned, you don't often need to *do* modeling any longer, you already know how to think about the complexities, you are, as it were, internalizing a "model in the mind" that may or may not be closely related to UML (I find it is not!). "*

The same respondent further comments that

"Who actually uses UML after the software has been implemented, when, and for what purposes?"

An author named Cliff Berg responded that

"in my book High-Assurance Design, I define almost 50 design patterns, and none of them required UML - yet I believe that the patterns are well defined, clear, and compact - much more compact than they would be with UML. I tend to always create diagrams for complex code, before I write the code. That is how I work. But the diagrams do get out of date. So often I used to see thick design documents with lots of class diagrams in them. What garbage."

In his another article [11] mentions (with reference to his expert friend)

"One of the differences between building architecture and software architecture is that a lot of decisions about a building are hard to change. It is hard to go back and change your basement, though it is possible."

It means that "Structural Engineers" have to do jobs more precisely than "Software Engineers". As, there is no way going backward (or iteration).

The above discussion leads us to think that though some practices such as Design and Project Planning in "SE" partially existed in some environments, yet, in essence "SE" has been a methodological subject. This view about "SE" is not possible to be modified. Due to all these misconceptions in a changing world and limitations of "SE" scope, we need to broaden the concepts which calls for a paradigm shift to visualize and practice the software development on a much larger scale. That view of larger scale is beyond just planning phase, requirement phase, design phase, testing phase, and maintenance phase and so on.

3. Paradigm Shift

Why a paradigm shift?

The research by [14] mentions the software community's focus constantly shifts because new force is applied when the last swing is failed. He suggests this is harmful. However, we, the authors, think this is not only positive but also unavoidable. For instance, the shift in software development processes since 1970's till to date (**Fig. 3**) is appreciated by the research and industrial community.

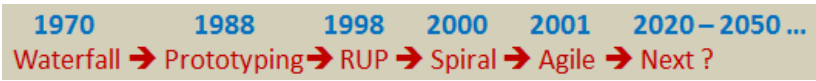

Fig. 3. Software development processes overtime

The research by [13] argues that computational science needs a new paradigm to address the prediction challenge. They specifically focus on validation and verification process and point out that most fields of computational science lack a mature, systematic software validation process that would give confidence in predictions made from computational models.

If we analyze the observations of the above two writers, we can see that several products and processes that were once popular one or two decades ago seem disappearing from the picture.

For instance, Yahoo! messenger and Yahoo groups, MSN messenger, AOL, Netscape, IE, Friendster, Outlook Express and may others. In "SE", shift in process happened constantly.

Coming back to our focused topic on "SE", according to [12] - Peter is the past president of ACM,

"We have not arrived at that point in software engineering practice where we can satisfy all the engineering criteria."
Our stance is that if after almost 45 years "SE" could not satisfy the engineering criteria then the field was misunderstood hence could not establish its own core concepts. Whereas many other IT related fields are already well established. This is calling for a paradigm shift.

The question is "Why the paradigm shift is happening?" The potential answer is that software development has gone beyond the boundaries of software development itself. It has involved many other domains which are beyond the scope of "Engineering" or "SE" or software development lifecycle. The new paradigm has worked like a Climate Change and we are now facing the disruptive weather conditions which current "SE" cannot respond. The new paradigm has started involving social coding, self-service or self-integration apps, mobile apps, collective intelligence in social networking, social psychology, customer and team collaboration, buzz of codeless future, mobile end-users, self-service capabilities to the business users. With these and more the need for reusability, intelligence, quality and so on has been multiplied.

A new paradigm which could visualize the software development on a broader and larger scale, forecast dynamic problems, guide teams and organization in disruptive situations and satisfy the human being needs as soon as possible. This view needs as we view the Climate Change and the solution could be hidden in Vibrant Softology concept.

4. Vibrant/Dynamic Softology (VS or DS)

The Vibrant/Dynamic Softology (VS or DS)

"The dynamic way of working to produce constantly changing, alive and stable software to satisfy the human and business centrism"

Terms explained:

Alive (A): Flexible, Adaptive (to change in need, tech), Responsive, Eventful, Uncertain, A range of regular and irregular behaviors, may be random or non- random
Stable (S): Rules, Conditions
Centrism is a political ideology based on reason and pragmatism considerate of short and long term thinking - Centrism is not defined by compromise or moderation, it is considerate of them.
Human-centric (H): Useful for human, Stakeholders, Interactive, Boundary, Culture
Business-centric (B): Business Services, business-focused project leadership, Shelf-life of applications

In a simple equation, ➔ VS = change (all); ➔ VS = change (A, S, H, B, *)

Where, 'all' denotes software products, processes, technology. Where, 'change' denotes vibrant. Where '*' denotes any other attributes to be added in future.

4.1. Analogy of Climate Change for Software Development Environment

Climate takes a long-term view whereas weather takes a short-term view. Climate tells you what wardrobe to have, whereas weather tells you what to wear on any given day.

Weather activities are dependent on the climate guidelines. However, we are directly and immediately affected by the weather rather than the climate. Our weather affects our choices we make on daily basis about whether to walk or take the car, what clothes we wear and whether outdoor events and pursuits are likely to get glorious sunshine or be rained off. We do not plan and prepare all these things in advance every morning in *systematic and disciplined ways*, because it is neither wise nor useful to do so rather time consuming and costly. The environment of Software creation is like climate and weather conditions. However, there are two exceptions to this analogy.

First exception: Significant natural climate change is realized in 30-50 years, whereas software climate change is realized in 3-10 year.

Second exception: Natural weather is dependent on seasons. The seasons come in a sequence one after the other such as spring, summer, autumn, winter, or monsoon. Whereas, the weather in software environment is not dependent on seasons. The seasons do not change in a sequence (systematically) in software environment rather any season can replace the other season-anytime – it means sequence is disruptive.

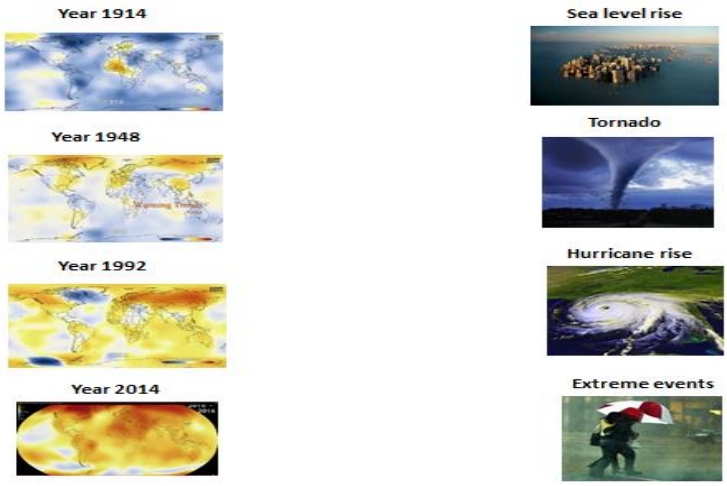

Fig. 4. Climate change and its effect on weather (1914-2014) – [Source: Nasa.gov]

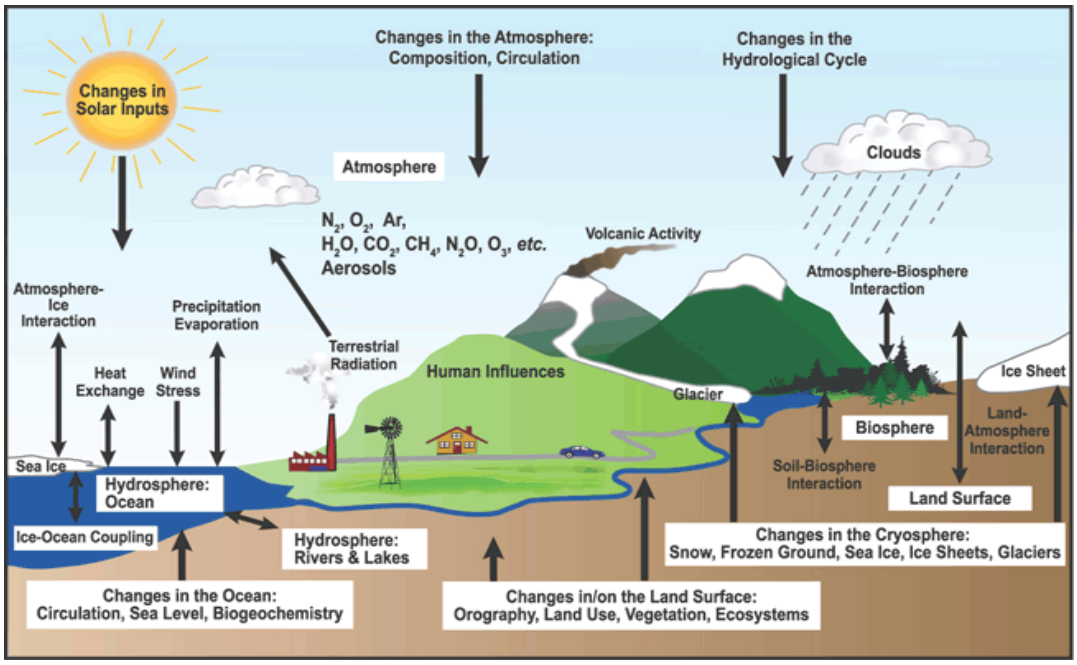

Fig. 5. Schematic view of the components of the climate system, their processes and interactions. [Source co2now.org]

By the term "software climate change", we mean that though it is commonly believed that Software has longer life than that hardware, however Software processes, products and range of its users has been constantly changing. As mentioned earlier above that several Software products and processes that were popular one or two decades ago seem disappearing from the environment. Some examples of such products are Yahoo! messenger and Yahoo groups, MSN messenger, AOL, Netscape, IE, Friendster, Outlook Express and may others. Some of the products were very short lived such as Microsoft Windows Millennium (ME), Microsoft Kin, HP iPhone and so on. The disappearance of popular Software happens because after using certain Software users' expectations are amplified (naturally). Due to this, Software climate collects new forces (cyber-psychology, technological growth, finance, global changes in business, security and so on) and replace the old ones – hence a climate change starts to take place. This climate change decides which software product will live and which should die. We can say the software climate forces are not controllable, whether software weather forces are controllable by us. The software climate forces work like a black-hole, collecting external forces, processing their feedback and then outputting the guidelines for controllable forces.

In order to understand this phenomenon, we have created a "Black-hole model".

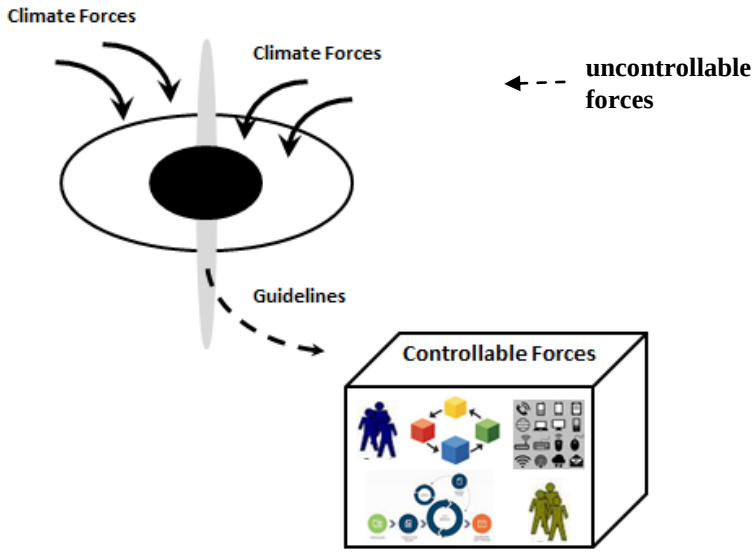

Fig. 6. Black-hole model – controllable and uncontrollable forces

In this discussion, our focus is to understand the life of our software product, or process or even a domain. This is important to know. This is discussed in the following section.

4.2 Predicting the Life of Software Product, Process or Domain

There is a question that "is it possible for us to predict the life of our software (though not accurately but adequately)?"

In order to understand and predict that what is the expected life of a Software product, process or a domain (which is a combination of concepts, processes and products such as social media domain), we have come up with a simple equation. The assumption is that, prior predicting the life expectancy of Software, we should have its past three (3) years of its perception of acceptance – at minimum. Ideally, we should have its past four (4) years of perception of acceptance. Here, perception means "what is the opinion of the users of a particular Software?" The results of our equation are based on subjective data, so the authors do not guarantee of complete accuracy, though the result would help to understand the probability of acceptance of Software in future. Also, geographical aspect is important. A Software product may be less popular (or even unknown) in Malaysia but may be very popular in Korea (such as Naver search engine is very popular in Korea, but almost unknown in Malaysia). So, it depends where the survey about perception has been conducted. For world-wide prediction a world-wide survey should be conducted using social media (Linkedin, Facebook, Twitter or other means).

$$\text{Software Life} = (oHW * pSW) * 2$$

Where, oHW denotes to optimal hardware support. If Software has highest optimal support by hardware, then the value of 0HW =1. As the hardware support goes down then value of oHW also goes down such 0.9, 0.8…..0.1. If value of oHW reaches 0.0 this means Software has not support by hardware hence cannot be executed – instant death of Software product. In case of predicting the life of a process the value of oHW may not be needed (such as UML or agile process does not necessary need hardware). This means either remove oHW or assign value of 1 to it.

Where, pSW denotes to Perception of Software acceptance. In other words, this is about the satisfaction of its users. This means that in the year of future prediction, what is the opinion of the active and non-active users about a particular Software. The Scale should be from 1 to 10. Where 1 is the minimum and 10 is the maximum. This should be done in a survey and try to obtain as much feedback about that Software as possible then calculate its mean.

Let's analyze the facts about a few popular Software to know what is the life expectancy of a Software from this year 2015.

MSN messenger 1995, Yahoo! Messenger (1998), Google+ (2011), Agile Process (2001), and UML (1997).

MSN Messenger Life = ***(1 * 2) * 2 = 4 Years***

Based on this calculation MSN Messenger will be alive for the next 4 years. It will be discontinued after that (unless new major innovation is introduced into it)

Yahoo! Messenger Life = ***(1 * 3) * 2 = 6 Years***

Based on this calculation Yahoo! Messenger will be alive for the next 6 years. It will be discontinued after that (unless new major innovation is introduced into it

Google+Life = (1 * 3) * 2 = 6 Years

Based on this calculation Google+ will be alive for the next 6 years. It will be discontinued after that (unless new major innovation is introduced into it

UML Life = ***(1 * 4) * 2 = 8 Years***

Based on this calculation UML will be alive for the next 8 years. It will be discontinued after that (unless new major innovation is introduced into it

Agile Methods Life = ***(1 * 7) * 2 = 14 Years***

Based on this calculation Agile Methods will be alive for the next 14 years. It will be discontinued after that (unless new major innovation is introduced into it

Based on the observation of appearing and disappearing of concepts, products and processes the Software climate change takes place between 5-20 years to collect forces that form the weather and set the destiny of Software, its teams and organizations.

4.3 Weather Analogy for Software Development Environment

The Software teams and organizations act or react according to the weather conditions. If weather is normal, they still work in a systematic and disciplined way (this means that "SE" has not been eliminated altogether; instead it is in a transition phase to the next level or new paradigm); if weather is not normal (but manageable) then they work in semi- systematic and semi-disciplined way; and if the weather is turned into severe (which is the case due to our current Climate) then teams do not work in systematic and disciplined way; they work according to the changing conditions; for instance website hacked or urgent requirement by the client on Saturday morning.

Natural climate change and weather conditions are constantly monitored. Data is collected from a number of sources — radar, observation stations, weather balloons, planes and satellites, and a network of 290,000 volunteer storm spotters — and then fed into vast mathematical simulations that churn out detailed local forecasts of what may happen in a few days or in a few hours' time. This is quite effective approach (so far).

In Vibrant Softology, we need to plan the similar strategies to cover the business trends, human digital psychology, finance technological growth, security and so on as shown in **Fig. 5**. Because, the software teams work in a disruptive environment so they need to respond to those conditions all the time.

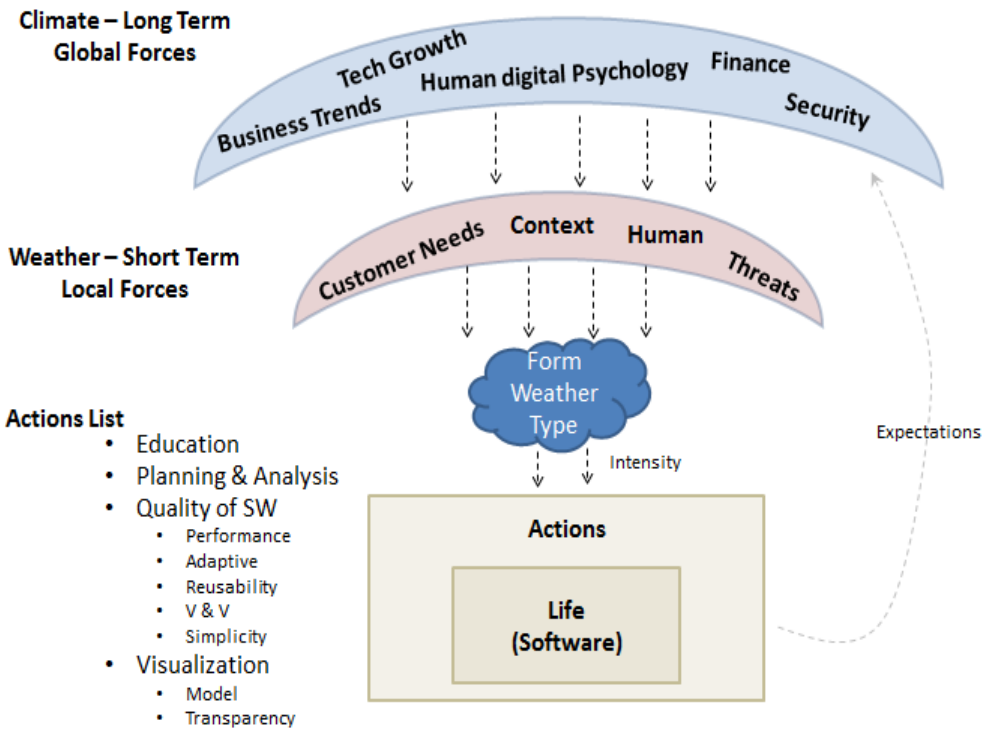

Fig. 7. Climate and weather analogy of Software development environment - Vibrant Environment

An interesting commonality of this analogy is that the natural earth climate has started getting affected by earth itself. Similarly, software development environment has also started getting affected by software itself – mostly the processing power (efficient algorithmic approaches) of the software has multiplied the expectations of the business-minded world.

Weather is never constant anywhere. However, in certain regions the temperature does not vary so much with the change of seasons throughout the year.

Hot weather regions: Extreme hot weather can be seen in the equatorial regions of the earth
Cold weather regions: Cold weather can be observed in the polar climate zones
Coastal weather regions: Coastal weather can be observed in the coastal or oceanic regions.
Mountain weather regions: The weather in the mountainous regions

The weather region analogy can be viewed as different teams working on different types of projects. Each, project according to the nature of project, can be placed in its relevant region, cold project (less active/ nor or a few changes), hot (constant changes) and so on. Each region can face different weather at any time– normal weather, stormy weather, tornado, and flood and so on. There could be many contributing factors to form a typical weather in a region. Overall, different weather conditions are happening on planet earth simultaneously. Similar situations happen in software environments of different organizations.

4.4 A Mindset Shift from "Software Engineering" to Vibrant/Dynamic Softology

Human Progress factor: We always leave behind the traditional way of working. We are leaving the phase of a disciplined way of being thorough, careful, and thoughtful, with that the notions need to transform accordingly. We want to develop software in easy, fast and simple way, and yet the software should be correct, its coverage should be broad and large scale. As mentioned above, Paradigm shift is happening hence there is a need to visualize the software development on a much larger scale. That view of larger scale is beyond just requirement phase, design phase, testing phase, maintenance phase and so on. A glimpse is illustrated in **Fig. 8**.

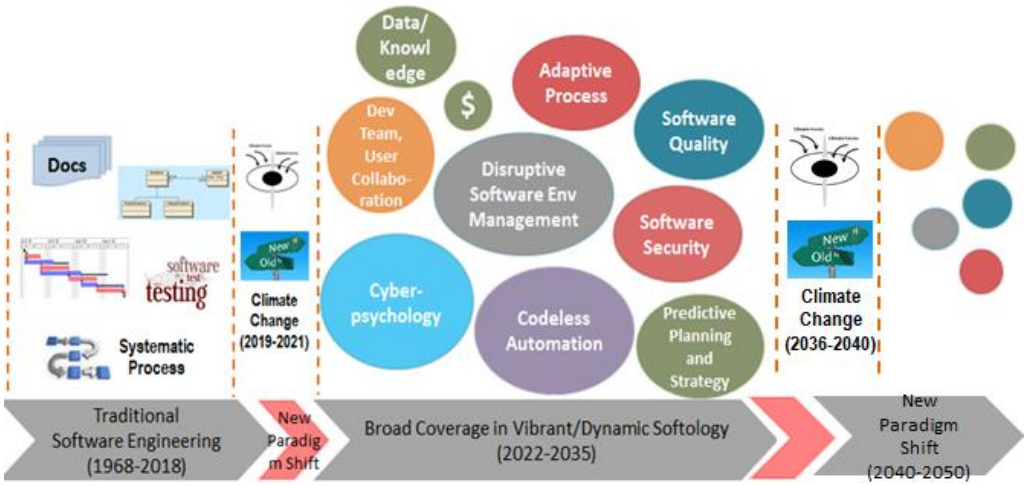

Fig. 8. Broad domain coverage in VS

The traditional "SE" focuses on developing systematic processes, heavy documentation, many types of diagrams, detailed planning, and testing. Whereas the new paradigm suggests to focus on cyberpsychology (human digital psychology, human digital identity, digital positive/negative relationships, personality types in cyberspace), development of adaptive processes, codeless automation, software teams, customer and end-user collaboration (fast feedback, globalization factor), prediction of requirements, big, fast, heterogeneous data, knowledge creating and so on. It means that the leading role of VS is to dictate or provide guidance to all other fields of IT. This is the umbrella of all other IT fields – which in fact "SE" supposed to had this leading role.. Notice the climate change strip in the above **Fig. 8**. This would bring new thoughts and expectations to software development community – the "Softologists".

5. Conclusion

It is appropriate to mention that his article is not meant a struggle to find a replacement of SE term or domain, rather focuses on to submit the transformation need in the field. We argue that that the rules of *systematic and discipline* proved effective for one discipline not necessarily result useful in the other. The analogy of the proposed idea is that most scientists are predicting that climate change will result in more severe weather; so we expect the same for software environments. We are argue that the traditional SE focuses on developing systematic processes, heavy documentation, many types of diagrams, detailed planning, and testing. Whereas, the new paradigm shift which is unavoidable due to external forces (creating black holes), suggests to discard some of these practices (or even sub-domains) and bring new domains into focus. Such as, the new focus would be on cyberpsychology (human digital psychology, human digital identity, digital positive/negative relationships, personality types in cyberspace), development of adaptive processes, codeless automation, software teams, customer and end-user collaboration (fast feedback, globalization factor), prediction of requirements, big, fast, heterogeneous data, knowledge creating and so on. Therefore, we argue that SE discipline in its current form has weaknesses and for many IT related experts is rather misleading. So, there is a need to refine the SE discipline beyond the "engineering mind-set".

References

[1] Johnson. P, Ekstedt. M and Jacobson. I, "Where's the Theory for Software Engineering?," *IEEE Software*, published by the IEEE computer society, 2012. Article (CrossRef Link)

[2] Kniberg. H, "Culture over process," *Article blog*, 2013. Article (CrossRef Link)

[3] SEMAT working Areas. Article (CrossRef Link)

[4] Sommerville, I, "Software Engineering," 9th Edition, *Pearson*, 2011.

[5] Juan. C. N, Carl. J, "Surfing the Edge of Chaos: Applications to Software Engineering," *Naval Postgraduate School Monterey Ca*, 2000.

[6] Perrow. C, "A Framework for the Comparative Analysis of Organizations," *American Sociological Review*, (32:2), pp. 194-208, 1967. Article (CrossRef Link)

[7] Burton, R. and Obel, B, "Strategic Organizational Diagnosis and Design. Developing Theory for Application," *Kluwer Academic Publishers*, 2nd Edition, 1998. Article (CrossRef Link)

[8] Petroski. H, "Design Paradigms: Case Histories of Error and Judgment in Engineering," *Cambridge University Press*, 1994. Article (CrossRef Link)
[9] Fowler. M, "YAGNI," May 2015. Article (CrossRef Link)
[10] Fowler. M, "Is Design Dead?," May 2004. Article (CrossRef Link)
[11] Fowler. M, "Who Needs an Architect?," *IEEE SOFTWARE Published by the IEEE Computer Society*, 2003. Article (CrossRef Link)
[12] Peter J. D and Richard. D. R., "The Profession of IT: Is Software Engineering, Engineering?," *Communications of the ACM*, Vo l. 5 2, No. 3, 2009. Article (CrossRef Link)
[13] Douglass E. P and Lawrence G. V, "Computational Science Demands a New Paradigm," *Physics Today*, pp 35-41, Vol 58, Issue 1, 2005. Article (CrossRef Link)
[14] Margaret J. D., "Process and Product: Dichotomy or Duality," *ACM SIGSOFT Software Engineering Notes Homepage archive*, Volume 20 Issue 2, Pages 17-18, ACM New York, NY, USA, April 1995. Article (CrossRef Link)

Imran Ghani is a Senior Lecturer at School of Information Technology, Monash University Malaysia, Sunway Campus. He received his Master of Information Technology Degree from UAAR (Pakistan), M.Sc. Computer Science from UTM (Malaysia) and Ph.D. from Kookmin University (South Korea). He is the member of Software Engineering Research Group (SERG). He has more than 80 publications in Journals, Proceedings, and Book Chapters.

Seung Ryul Jeong is a Professor in the Graduate School of Business IT at Kookmin University, Seoul, Korea. He holds a B.A. in Economics from Sogang University, Korea, an M.S. in MIS from University of Wisconsin, and a Ph.D. in MIS from the University of South Carolina, U.S.A. Dr. Jeong has published extensively in the information systems field, with over 80 publications in refereed journals like Journal of MIS, Communications of the ACM, Information and Management, Journal of Systems and Software, Enterprise Information Systems, Online Information Review, among others. Dr. Jeong's areas of interest are Software Engineering, Systems Implementation, Opinion Mining, Process Management, and Information Resource Management.