ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

Generalized Software Techniques

Piyush Singh Katiyar
Assistant System Engineer, Tata Consultancy Services Ltd., New Delhi, India

Abstract
Software has become a more important component of computer systems. However, universal methods of software engineering have yet to be established. Despite the rise of software, challenges endemic to the software engineering process as well as apprehension to any change in the process have made it difficult to establish universal techniques. However, a lack of structured software development processes likely leads to higher production costs and other negative consequences. In this paper, barriers to the implementation of universal software methods will be discussed. Additionally, support for formalized software techniques and their ability to deal with the issues in software development will be given and justified.

Keywords
Universal Software Technique, Formal Software Methods, Search-Based Software Engineering Algorithms.

I. Introduction
First, it is vital to discuss what is meant by a universal software technique as well as why it is difficult to put in place. This term refers to a systematic, structured system that can be relied upon to create code that is reusable, scalable, generalizable, and include descriptions of "high-level behavior and properties of the system".

II. Universal Software Technique
As it stands now, it is challenging to implement universal methods as it does not fit well with the software engineering process. Specifically, a significant amount of programmers' time is spent updating the code to reflect changing requirements and specifications. This circular process where code is submitted than changed is not in accord with a sequential procedure.

III. Apprehension
Another barrier to the widespread use of software engineering methods is apprehension. Software engineers tend to view them as only being suited for specific projects and purposes rather than seeing them as a staple in the software engineering process. Further, there is a tendency to underestimate the technological advancement certain methods can bring to a software project.

IV. Abstract Software Design Process Barriers
The abstract nature of the software design process also makes it difficult to implement specific software methods. As J. Jones mentions in his 1970 book Design Methods: Seeds of Human Future, designers are charged with foreseeing what the end product will be and creating a viable product out of that vision. This is a rather impossible task with the frequent adjustments of software specifications and requirements. Don Batory, a prominent computer scientist currently teaching at the University of Texas, argues for the formation of a science of design. In his view, the abstract nature of design is one of the pitfalls of software engineering. Furthermore, lack of communication and, to an extent, mutual understanding between designers and programmers lead to programmers having to make design decisions. As a result, many attempts at creating defined techniques has resulted in methods that lack structure and do not have a specific purpose assigned to each step.

V. Formal Software Methods
However, there have been attempts to introduce formal software methods. Some examples include object-oriented and data-flow models. In the object-oriented model, all components are treated as separate objects rather as one uniform system. Objects can request data and services from other objects. This model has become widely used in modern software development. The data-flow model has been implemented into programs like C and Pascal with some success. Other scientists prefer developing more research oriented models based on psychology through the use of controlled experiments, case studies, and surveys.

VI. Money Management
In order to advocate for the implementation of software methods, areas in which in these methods can improve the software engineering process will be identified. One major area is that of money management. For purchasers, receiving a product at the lowest cost possible is of the utmost importance. However, if the average programmer can only produce 15 lines of code a day, keeping overhead low is challenging. Moreover, this problem is exacerbated by the revisions needed before releasing a final product.

VII. Timeliness
Another area that can be improved with the use of software methods is timeliness. Purchasers want their product promptly. Additionally, even if the company is producing the software for its own use, deadlines must be met. However, the length and inefficiency of the revision process makes this problematic. Moreover, with a structured process in place, it is likely that the ease of maintenance and revision will be increased as the code becomes more uniform.

VIII. Automatic Code Generator
As several areas that could be enhanced through formal methods have been identified, it is important to discuss how certain methods could improve performance. The Earned Value Management method discussed in Boehm's 2003 paper attempts to create a concrete breakdown of the different steps of a software project and the monetary cost involved. Every step is given a proposed budget. The total projected cost is called the "budget cost of work scheduled." By comparing this to the budget cost of work performed, one is able to better manage project costs.
The automatic code generator advocated by Joseph is a technique that would likely improve the issues of timeliness and maintenance. As mentioned earlier, programmers average about 15 lines of code per day, making the construction and revision progress costly and lengthy. Since many software programs have anywhere from 1 to 5 million lines of code, it can take years for a team of 100 to be complete an application. In contrast, a team of 80 to 90 developers were able to use automatic code generation through UML (Unified

Modeling Language) models to produce 6 million lines of code in 6 months. Moreover, the resulting code is uniform, eliminating concern of code compatibility. As revising one section of code usually requires updating other sections which interface with the code, automatic code generators provide a way to minimize time spent on this operation. Additionally, generated code is reusable making it easier to update the application in the future. Furthermore, automatically generated code is also scalable and generalizable, making it an excellent tool for software projects of various kinds.

Other formal techniques have been introduced and supported by software engineers. These techniques will be examined and contrasted with automatic code generation for better analysis. In his 2003 paper, Boehm recommends having software and systems engineers work together. He argues that this will allow for project requirements and project code to be developed simultaneously. The main benefit would be to decrease the amount of time spent in revision. Though this method would help engineers cope with unscheduled revisions and software incompatibility, the speed of code production is not affected. Additionally, this technique relies on communication and teamwork to create uniform code rather than it being an instrinsic part of the method as it is in automatic code generation. Furthermore, Liu and Venkatesh point out in their 2008 paper that the lack of clear separation between requirements and implementation makes it difficult for architects and programmers to work together. As a result, Boehm's approach might do more harm than good for productivity for a project.

IX. Search-Based Software Engineering

Another useful software technique is search-based software engineering (SBSE). In this approach, the process of software engineering is seen as a series of optimization problems. Using tools like genetic algorithms and fitness functions, a search is implemented in the search space, a representation of all possible solutions to a problem. Once the algorithm finds the best solution, the search terminates.

Like automatic code generators, these algorithms are scalable and generalizable. Parallel execution allows for multiple searches to be carried out at once making SBSE a reasonable tool for software applications of any size. Additionally, SBSE only requires a representation of the program and a fitness function designed to target the optimal solution to the problem, allowing for this method to be used in software of many types.

Two examples of SBSE are the Hill Climbing and Simulated Annealing algorithms. The Hill Climbing algorithm randomly chooses a point in the search space. It then searches for better solutions in the nearby search space, selecting the optimal result in that area. This process continues until the algorithm finds the best solution in the search space. In contrast, the Simulated Annealing algorithm works through a "temperature" based method. When the temperature is high, the algorithm is open to selecting all choices, even poor ones. As the temperature decreases, the better choices are favored more often. This continues until the freezing point is reached at which the optimal solution is chosen. One will yield the same result regardless of whether Hill Climbing or Simulated Annealing is chosen.

X. Conclusion

Software engineering methods could be extremely useful and beneficial for both companies and consumers alike. Despite the apprehension and challenges of introducing structure into software engineering, doing so would likely bring the benefits of higher profit and more efficiency to the companies who chose to do so. Moreover, as software becomes a bigger component of computer systems, such changes will likely become necessary to cope with the growth of the software industry.

XI. Acknowledgement

I would like to thank Mr. Virendra Kumar, Assistant Professor, School of Computing Science & Engineering, Galgotias University, for his most support and encouragement. He kindly read my paper and offered invaluable detailed advices on grammar, organization, and the theme of the paper.

References

[1] *Babb, R. (1984). Tools for Large Scale Software Engineering. Oregon Health & Science University, 11-11.*

[2] *Boehm, B. (2003). Value-Based Software Engineering. Software Engineering Notes, 28(2), 1-7. Retrieved November 25, 2014, from http://www.csee.umbc.edu/~sweet/Ph.D/papers/VBSE/VBSE-Boehm.pdf*

[3] *Easterbrook, S., Singer, J., Storey, M., & Damian, D. (2008). Selecting Empirical Methods for Software Engineering Research. In Guide to Advanced Empirical Software Engineering. London: Springer.*

[4] *Harman, M., McMinn, P., De Souza, J., & Yoo, S. (2012). Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In Empirical Software Engineering and Verification. London: Springer.*

[5] *Introducing Formal Methods. (2002). Retrieved November 25, 2014, from http://web.mit.edu/16.35/www/lecturenotes/FormalMethods.pdf*

[6] *Jones, J. (1970). Design Methods: Seeds of Human Future. New York: Wiley & Sons.*

[7] *Joseph, M. (n.d.). Formal Techniques in Large-Scale Software Engineering. 1-7.*

[8] *Liu, Z., & Venkatesh, R. (2008). Methods and Tools for Formal Software Engineering. In Verified Software: Tools, Theories, Experiments (p. 31, 33). London: Springer.*

[9] *Software Engineering Values. (n.d.). Retrieved November 25, 2014, from http://www.d.umn.edu/~gshute/softeng/values.html*

[10] *Telkar, N. (2010). Software Design Techniques. Retrieved November 25, 2014, from http://www.cs.colorado.edu/~kena/classes/5828/s10/presentations/softwaredesign.pdf*

[11] *Torp, K. (2002). Software Engineering Techniques. Retrieved November 25, 2014, from http://people.cs.aau.dk/~torp/Teaching/E02/OOP/handouts/design_patterns.pdf*

Author's Profile

Mr. Piyush Singh Katiyar (Assistant System Engineer), His Educational Qualification is: B.Tech. (Computer science and Engg.) from Integral University, Lucknow, India. Current Employer is Tata Consultancy Services Ltd., Gurgaon, India. His working area: Software Engineering and Cloud computing.