International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 06 | June 2020 www.irjet.net p-ISSN: 2395-0072

Decomposition and Modularity in Software Systems

Anmol Gaba[1], Dr. G N Srinivasan[2]

[1]Department of Information Science and Engineering, RV College of Engineering, Bengaluru, Karnataka, India
[2]Professor, Department of Information Science and Engineering, RV College of Engineering, Bengaluru, Karnataka, India

---

Abstract - *This paper ventures into the discussion about software decomposition and the related modularity aspects necessary for the development of modern software architectures. There has been an evident rise in the efforts by organizations to develop and master the standard practices, be it agile or the regular waterfall model. In relation to this , modern programming frameworks need to develop consistently so as to abstain from getting less helpful, as it is in the case of many legacy systems riddled with the technological debts. Be that as it may, rehashed changes in the product may block the internal nature of the framework. Seclusion of the program modules is viewed as a significant part of a decent internal quality, and the useful decomposition is a methodology that empowers to accomplish great particularity. Nevertheless, existing methodologies for useful deterioration overlook implementation endeavors, and this may cause a circumstance where the necessary changes are too exorbitant to even think about implementing. In this paper we portray a way to deal with useful deterioration for programming design advancement considering additionally the usage endeavors.*

Key Words: Software Modularity, Decomposition, Legacy Software, Modern Systems, System Architecture

1. INTRODUCTION

Programming frameworks must develop after some time, else they continuously become less helpful. When a framework is discharged, it constantly changes, for example because of developing client prerequisites (perfective changes), bug fixes (restorative changes), stage modifications (versatile changes), or revision of inactive blames before they become operational issues (preventive changes). Thus, the framework floats away from its underlying engineering because of the developmental changes.

A significant part of decent programming engineering is its measured quality. A framework may at first comprise exceptionally durable subsystems with low coupling between them. Be that as it may, as greater usefulness is added to the framework, their coupling will in general increment and their union abatements. Consequently the framework turns out to be less reasonable for engineers, bringing about diminished quality and a framework that is progressively hard to keep up.

Programming engineering is the significant level plan and structure of the product framework. While planning the framework is speedy, reasonably contrasted and building the framework, it is basic to get the design right. When the framework is constructed, if the engineering is damaged, wrong, or only deficient for your necessities, it is amazingly costly to keep up or broaden the framework.

The substance of the engineering of any framework is the breakdown of the idea of the framework all in all into its including parts, be it a vehicle, a house, a PC, or a product framework. A decent engineer likewise recommends how these segments collaborate at run-time. The demonstration of recognizing the constituent parts of a framework is called framework decay.

2. LITERATURE SURVEY

In [1] authors find that in the plan advancement of programming improvement measured quality has consistently had a solid effect. It is seen that despite the fact that firmly coupled parts are more enthusiastic to keep up; they have a high probability of endurance and use in resulting adaptations of a planned worldview. Despite the fact that they streamline advancement, they experience more shock changes to their reliance connections that are not related to new usefulness. They become "more enthusiastic to increase," in that the new segments included in every rendition are more secluded than the first heritage structure.

In [2] it is seen that long-running programming frameworks experience the ill effects of programming disintegration, because of their consistent advancement to meet new or evolving necessities, seriously constraining their viability. Relocating programming frameworks, for example moving heritage frameworks into present day conditions and advancements without evolving usefulness, is a key strategy of programming development, and serves to continue existing programming frameworks operational. Organized movements take into consideration moving set up programming answers for bleeding edge innovation, without thinking about the fundamentally

higher dangers of building up another framework without any preparation.

In [3] authors examine the procedure of ceaseless support movement and its motivation: the wellsprings of transformative weight on computer applications and projects. It proceeds to characterize programs as indicated by their relationship to nature in which they are executed. The existence cycle forms are likewise quickly talked about. Prologue to laws of Program Evolution defined after the quantitative investigations of the development of various frameworks is made. At long last a model is given which shows the utilization of Evolution Dynamics models and the connection to program discharge arranging.

In [4] the paper portrays a way to deal with functional decomposition for programming design development considering additionally the execution endeavors. There is a requirement for consistent advancement of programming frameworks so as to abstain from losing use. Be that as it may, persistent changes in the product may prompt annihilation of the framework quality and helpfulness. Seclusion is viewed as a significant angle to accomplish a decent programming quality, the useful disintegration is a methodology that accomplishes it better. All things considered, existing methodologies for useful disintegration may make the necessary changes the product too expensive to even think about implementing, as they regularly will in general overlook the hidden usage endeavors.

The discussion of the paper in [5] talking about a PhD research examine venture tending to many key difficulties concerning Microservice Architecture (MSA) lead by the accompanying variables: sway on the procedure of movement of the current applications towards MSA, examination on a depiction language for structuring and investigating designs, the key properties of microservice models. The essential commitment is precise mapping to concentrate on architecting microservices acted so as to comprehend the ebb and flow condition of the exploration and the fundamental potential holes in the region.

A few measurements discussed in [6] for every one of five sorts of programming quality measurements: item quality, in-process quality, testing quality, upkeep quality, and consumer loyalty quality are assessed. The primary commitment of this work is the simple and extensible answer for the programming nature of approval and confirmation in the product improvement process. In this way, we utilize formal methodologies so as to depict the essential parts of the product. This formalization bolsters the assessment of the measurements or estimation level themselves.

3. METHODOLOGY

The following methodology is followed for decomposition:

1. One method of performing software decomposition is to have the same number of administrations as there are varieties of the functionalities. This disintegration prompts a blast of administrations, since a conventionally estimated framework may have several functionalities. In addition to the fact that you have an excessive number of administrations, yet these administrations regularly copy a great deal of the basic usefulness, each tweaked to their case. The blast of administrations perpetrates a disproportional expense in combination and testing and expands generally speaking unpredictability.

2. Another software decomposition approach is to lump every single imaginable method of playing out the activities into super administrations. This prompts swelling in the size of the administrations, making them excessively intricate and difficult to keep up. Such god stone monuments become an appalling dumping reason for every single related variety of the first usefulness, with perplexing connections inside and between the administrations.

3. The third approach, offering a basic yet incredible strategy for breaking down a framework, functional class disintegration (FCD) produces a design that is more strong than customary item situated deterioration for a few programming building assignments. A half and half technique that incorporates organized examination with an OO approach, FCD recognizes classes in corresponding with breaking down the framework into a chain of importance of useful modules. As of late, engineers stretched out FCD to incorporate UML ideas. Valuable for apportioning a framework for dispersion, the FCD pecking order gives a system to control improvement in a conveyed programming building condition. It likewise recognizes and coordinates segments in segment based turn of events and supports the framework life-cycle upkeep stage.

There are numerous other methodologies or techniques to modularize the system. A single method or a combination of these can be undertaken in the initial stages of the program itself so as to build out the system according to the planning commissioned by the development team.

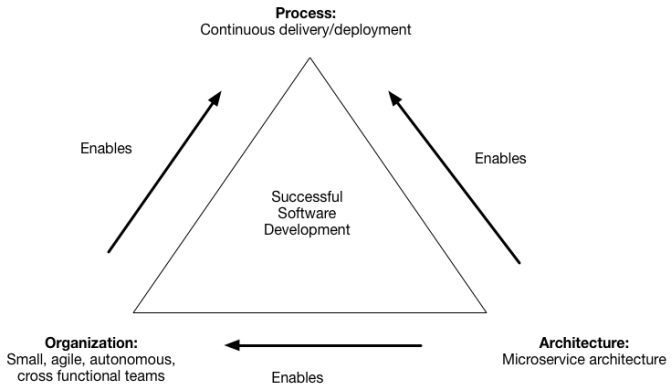

Fig -1: Software Development Process

Exceptionally coupled are free or practically autonomous. Modules are autonomous in the event that they can work totally without the nearness of the other. The more associations between modules, the more reliant they are as more information around one module is required to comprehend the other module. In general, modules firmly coupled on the off chance that they utilize shared factors or on the off chance that they trade control data. Free coupling if data held inside a unit and interface with different units by means of parameter records. Tight coupling is needed whenever information is shared globally.

Cohesion is a measure of how well modules fit together. A component should implement a single logical function or single logical entity. All the parts should contribute to the implementation. Temporal cohesion, Procedural cohesion, Communicational cohesion, Sequential cohesion, Informational cohesion, Functional cohesion are the different types possible.

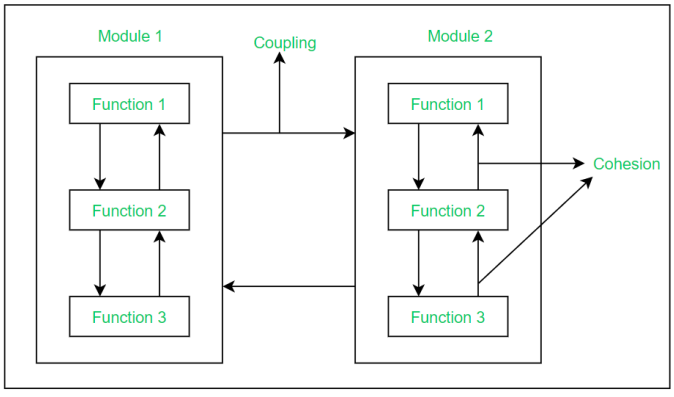

Fig -2: Cohesion and Coupling in modular programming

4. RESULTS AND DISCUSSIONS

The following points are realised in this report from the extensive studies of the related work:

- The architecture built must be well researched and stable for a couple of software generational changes

- Modules ought to be loosely coupled: each service as an API must encapsulate its implementation. The implementation can be changed without affecting the end users directly.

- Services (or modules) must be strongly cohesive: they should implement a small set of functions related strongly to each other in meaningful ways.

- A module should be testable: unit testing and integration testing at the very least of it.

- Services and modules must conform to the Common Closure Principle: it states that things that change together must be packaged together. This is done to ensure that each change affects only one service and does not have the rolling over or the 'domino' effect on other modules.

- Each service is small enough to be developed by a small team of 6-10 people making management easy.

- Each team that owns one or more services must be autonomous. A team must be able to develop and deploy their services with minimal collaboration with other teams.

There are various concerns highlighted with the use of legacy software which need to be addressed:

1. The legacy system suffers from legacy architectural design issues on a software development level and associated technical debt.

2. Existing APIs fetch more from the GET calls than what is required, leading to the emergence of technologies like GraphQL instead of traditional ones like REST.

3. There might be an issue of scalability of the existing systems due to software scale limitations.

4. There might be system performance issues with the overhead calls that are commonplace in older coding styles.

5. The backend code is not optimised leading to increased downtime.

5. CONCLUSIONS

This study was conducted to determine the current state of software architectural techniques followed by the software development industries. The practices in order

and newer techniques were outlined along with their purview.

From a developer's perspective there are mainly 2 advantages to improving a software's architecture and bringing it to modern standards:

1. Improved reliability, performance, robustness of the software.

2. Less downtime, more feature set delivery in lowered intervals.

This would also be a great advantage from the organization's perspective as:

1. Enhancement of the process of the monitoring of performance, errors, isolation of issues.

2. Development of more features takes place in the future more easily and the changes are put out more promptly.

REFERENCES

[1] MacCormack, A., Rusnak, J., & Baldwin, C. "The impact of component modularity on design evolution: Evidence from the software industry", Harvard Business School Working Paper. (2007)

[2] J. Jelschen, G. Pandey, A. Winter. "Towards Quality- Driven Software Migration". In Proceedings of the 1st Collaborative Workshop on Evolution and Maintenance of Long-Living Systems, Kiel, 2014.

[3] M. M. Lehman, "Programs, life cycles, and laws of software evolution," in Proceedings of the IEEE, vol. 68, no. 9, pp. 1060-1076, Sept. 1980.

[4] Faitelson, David & Heinrich, Robert & Tyszberowicz, Shmuel. (2018). "Functional Decomposition for Software Architecture Evolution".

[5] P. Di Francesco, "Architecting Microservices," 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, 2017, pp. 224-229.

[6] Lee, Ming-Chang. (2013). "Software measurement and software metrics in software quality". International Journal of software engineering and its application.

[7] Carl K. Chang, Jane Cleland-Haung, Shiyan Hua, and Annie Kuntzmann-Combelles. 2001. Function-Class Decomposition: A Hybrid Software Engineering Method. Computer 34, 12 (December 2001), 87–93.

[8] Cai, Yuanfang & Huynh, Sunny. (2020). Logic-Based Software Project Decomposition.

[9] Pang C., Szafron D. (2014) "Single Source of Truth (SSOT) for Service Oriented Architecture (SOA)". ICSOC 2014. Lecture Notes in Computer Science, vol 8831. Springer, Berlin, Heidelberg.

[10] Ghanam, Yaser & Carpendale, Sheelagh. (2008). A Survey Paper on Software Architecture Visualization.

[11] Bosch, Jan. (2004). Software Architecture: The Next Step. Software architecture. 3047. 194-199.

[12] C. G. Davis and C. R. Vick, "The Software Development System," in *IEEE Transactions on Software Engineering*, vol. SE-3, no. 1, pp. 69-84, Jan. 1977.