Reading Assignment - “Big Ball of Mud”

According to the article “Big Ball of Mud,” a lot of architectures have been introduced, such as “Pipeline” and “Layered” architecture. However, the most used architecture is “Big Ball of Mud.” The design of “Big Ball of Mud” is terrible, but we usually see it. This type of design makes the system out-of-order, and it may cause all the essential data to be the global or duplicated. It is very dangerous because global data are apt to be altered easily everywhere, and the duplicated data results in the waste of resource. These two situations show that the overall system with the architecture of “Big Ball of Mud” is not a well-designed architecture. Over time, the dirty portion will pollute and erode the whole system.

Unfortunately, this architecture has been lasting and prospering. In this article, the reason why “Big Ball of Mud” is popular and why good programmers allow to establish the terrible system are discussed. At the same time, the article also mentions how to prevent it and make the system better. The author goes through from “Big Ball of Mud,” “Throwaway,” “Piecemeal Growth,” “Keep It Working,” “Shearing Layers,” “Sweeping It Under the Rug,” and “Reconstruction” sequentially.

Sometimes, “Big Ball of Mud” comes from “Throwaway Code.” The purpose of “Throwaway Code” is only for fixing immediate issues or testing. Once used, it should be removed. The original issues and tasks need to be re-estimated because “Throwaway Code” is definitely the final solution. In my opinion, this is because some people always want to have an easy and convenient way to fix problems or complete the tasks. They don’t want to find the cause of the issue or consider more about all the details about the task. Over time, “Throwaway Code” is unconsciously reserved and then becomes “Big Ball of Mud.” For example, when I worked for a Taiwanese company, Compal. It is an original-design-manufacturing(ODM) company, and notebook, smartphone, and tablet are their major products. All the designs are located in Taipei, while manufacturers are located in Shanghai. I was a software project manager and was in charge of Lenovo’s Thinkpad tablet. The working model was that we needed to compile the testing image files in Taipei and brought them to Shanghai factory to manufacture. We always hoped that there were no issues in the production line. However, the thing usually did not run as your expectation. Once production issues blocked the line, engineers needed to handle it immediately. If engineers could not find the solution as soon as possible, the situation was that all the production managers would put pressure on them. Consequently, when engineers had a solution, they had to apply it to testing images right away. If the issues were fixed, the production would continue to proceed, and usually, no one wanted to re-think the cause at this moment. My job was to not only make sure the problems were fixed but also to ensure that our codebase should be clean without any temporary solution. At this time, I would require engineers to collect the log and data from the production line and send them to Taipei site for further analysis. Realizing the cause, generating the solution, and completing the full test are the proper approaches to avoid “Big Ball of Mud.”

Why does “Big Ball of Mud” prosper in the world? There are many “Forces” which direct the well-designed architecture to be “Big Ball of Mud.” The first reason is time. If people do not have enough time to think long-term impacts, short-term solutions will be accomplices which push the current system to “Big Ball of Mud.” It was totally proved in my previous work experience. No production manager wanted to give engineers enough time to think the long-term solution. All production managers needed was merely the short-term solution to pass the current blocked issue, and they could go home on-time. The tangible effect of short-term is that no critical issue block the production line. However, well-designed architecture does not easily reveal. Even though people benefit from it later, and the system is robust and durable, they still don’t understand that it is owing to well-designed architecture. “Architecture can be looked upon as a Risk, that will consume resources better directed at meeting a fleeting market window, or as an Opportunity to lay the groundwork for a commanding advantage down the road.” A good architecture could help the system prosper, but a bad architecture could erode the system and result in

devastation.

The second reason is cost. When people investigate into a new field, it will cost a lot. The earlier people put efforts into the system; the more people can benefit. It reminded me of my factory experience. If the test plan of the product project could be well-designed, and the consideration also included the future expansion, the total cost of this project could be greatly decreased comparing with a bad design. A detailed example is that in the engineering validation phase, EVT, we only needed to produce two hundred devices, and then we needed to yield three thousand devices in design validation phase, DVT. Finally, we needed to manufacture thirty thousand devices in production validation phase, PVT. If the engineer only considered the scale of two hundred devices to design the test plan, it might be no problem during EVT. However, when it came to the DVT phase, the original plan applied in EVT could be inappropriate due to different scales. In PVT, the scale was highly larger than DVT. The worst case was that the original plan should be thrown away, and maybe the production line should be re-allocated which cost a lot.

The third reason is experience. If people don't have enough experience in the field, it restricts the complexity of the system in architectural perspective, especially in the evaluation phase. Different people have different experiences. Consequently, the results generated by different people could be totally different. For example, there were two managers in my previous company. They were assigned to individually design a software architecture for Windows 8 App related to file management function. I joined their presentation and found that the architecture created by the manager with less experience was less efficient in scalability and integrability than the other one with more experience. This example was shown that experience limits the architectural sophistication of the system.

The fourth reason is skill. Different programmers may have different skills. Some programmers are talented in finding good abstraction, but some are good at driving people to the complicated codes which are hard to be understood. As to the preference of languages and tools, different people also have different choices.

The fifth reason is visibility. Programming is intangible, and only the programmer who creates the program is able to see the inside of the program. The way to present the logic of the software also affects how people understand it. Eventually, if the system is workable and has no problem, no one will care about the architecture of this system.

The sixth reason is complexity. If a system is too difficult to understand, it also means that this system is a complicated system. Legacy systems usually have their history, and the relations and boundaries between their components have been defined. Once systems need to satisfy new requirements, sometimes programmers will cross the boundaries. Consequently, big problems will arise because the original architectures are destroyed by these new implementations across the boundaries. This kind of damage makes systems get worse, and then systems become "Big Ball of Mud."

The seventh reason is change. Although the architecture is designed to satisfy the future requirement, people cannot exactly predict how the development is going. Once new requirement is against people's prediction, the change will slowly ruin the system. At this moment, the right way to fix this issue is to redesign the system.

The final reason is scale. Different scales should be managed in different ways. For example, a way to govern one million people should be totally different from twenty people. "Divide-and-Conquer" is not an inadequate answer to the scale problem. As a result, scale is also a challenge of preventing a system from becoming a "Big Ball of Mud."

Today, many software systems are similar with shantytowns. They are un-organized architectures. People do not invest sufficient efforts in building a well-defined architecture first. In the meantime, tools and infrastructures are inadequate. Some particular parts of this system are out of control and expand too much because there is no rule to regulate the entire software system. Finally, the whole system becomes chaos. However, this kind of system satisfies immediate needs because people do not care to have a perfect system in the beginning, and they only desire to have a program as soon as possible no matter this program includes a well-defined architecture or not.

Software can be changed due to its flexibility. “Because software is so flexible, it is often asked to bear the burden of architectural compromises late in the development cycle of hardware/software deliverables precisely because of its flexibility.” In my previous eleven-year work experience, I have suffered this misunderstanding a lot. My previous companies were both the ODMs, and they focused on hardware, and software was something belonging to their hardware products. Every time, when our software teams asked hardware teams to provide their requirements in the first phase, hardware teams always said they were busy on designing their circuits and mechanisms, and they did not have time to help with our software team. Over time, the software had already locked the architecture. Hardware teams started to provide their requirements and required us to modify our software. That’s because they always thought that software was flexible and could be altered anytime. Our supervisor also agreed to this idea because they didn’t have software backgrounds. This problem became serious when it was near the mass production. The layout and mechanism were able to change because they cost a lot of money and time, while software only needed to re-compile and re-download images. This is definitely a huge misunderstanding. If hardware teams can work with software teams from the beginning phase of the project, things will be expected to go smoothly, and there will be fewer troubles in mass production.

“The time and money to chase perfection are seldom available, nor should they be.” I deeply agree with this sentence. This situation should not happen because companies never offer this kind of profits and save money as well as time. Companies seldom have the patience to wait for a well-designed architecture because an excellent architecture usually costs a lot of resources.

“If a team completes a project with time to spare, today’s managers are likely to take that as a sign to provide less time and money or fewer people the next time.” I can’t agree any more with this idea. In my previous companies, when everyone was very busy with their current work, the supervisor would not ask too much, and all he wanted was the achievements. While programmers were not very busy and went home earlier without extra unpaid work hours, the supervisor would start to require individual functional-managers to provide the time-sheet reports, and then he would re-estimate the resource. Maybe he would re-allocate the resource or fire some non-productive programmers.

“You need to deliver quality software on time, and under budget.” This is always the principle in the software company. The supervisor required our managers to reach this achievement. The requirements he mentioned in the manager meeting were about how to decrease the budget and how to accelerate the schedule. However, the final achievements he desired to obtain were the same with ones which deploy less time and money.

“Architecture is a long-term investment. It is easy for the people who are paying the bills to dismiss it, unless there is some tangible immediate benefit.” An architecture will determine the future success or failure of a system. A good architecture will lead to a system prospering, and people who are in charge of this system will not need to worry too much. On the contrary, a bad architecture will result in chaos. It will take a lot of time filling holes or fixing the issues, and this situation might last until the end of this system. I assume that many programmers know this principle, but it is not often included in supervisors’ mindset. As a result, a bad circle occurs.

“On average, average organizations will have average people.” In average companies, if they have a hyperproductive architect who can create a complicated and efficient architecture for a system, I believe that he or she will require a lot of data or materials to design a well-defined architecture in the very early stage. In the meantime, other employees even managers and supervisors think it is not necessary to do this. Consequently, this hyperproductive architect cannot create his or her value and may choose to work for another company. Employee turnover becomes an issue when the companies do not realize how important a good architect is.

The scale of organization is also an important reason why architecture is ignored. Larger companies usually have more projects, cultures, processes, and conflicts than smaller ones, and organizational and resource allocation usually play the most important roles than other

factors. At the same time, technical concerns could be ignored, and architectures become less important.

“Therefore, focus first on features and functionality, then focus on architecture and performance.” This idea is usually rooted in managers and supervisors’ mindset, and they require programmers to follow. In most cases, programmers contribute their efforts to develop a system with full sets of features and functions in the beginning, but then this system becomes “Big Ball of Mud.” When looking the inside of the system, people will find that this system lacks a well-designed architecture, its names of variables and functions may be disordered and confusing, and the global variables and functions are widespread everywhere in this system. In addition, many parameters are included in one function, and the codes are duplicated. Another thing is that the code flow is too difficult to trace. When all the disadvantages combine, this system truly turns into “Big Ball of Mud.” It is extremely hard to focus on architecture and performance.

“Only by understanding the logic of its appeal can we channel or counteract the forces that lead to a Big Ball of Mud.” People need to know the reason why “Big Ball of Mud” is popular in the world first, and then they are able to avoid it. “Big Ball of Mud” is a kind of mode which people cannot deny. However, it is a passive attitude toward software development. In many cases, people are not willing to put many efforts and many resources in devising an excellent architecture first. Later, when the issues emerge, people just want to use patches to fill the holes. Consequently, signs of patches are everywhere, and these patches usually destroy the original architecture. Now, if people change their mind and act without passivity, they may be able to prevent “Big Ball of Mud.”

A top-down design may be a big problem. Some architects assert that this kind of design can make things better. However, this design is rigid. It causes a system to utilize resources inefficiently, and it is challenging in the analysis.

“Make it work. Make it right. Make it fast.” This means that people should let the system be workable and then make sure everything is on the right track without errors. The preparation should be done in the early phase. Later, performance could be focused after the architecture is already well-defined. Finally, reducing the cost is the main point after everything is done.

“Domain experience is an essential ingredient in any framework design effort.” The most efficient approach to get domain experience early is to find experts in the current field. At the same time, the tools that programmers use also affect architectures in systems. Due to different characteristics and limitations, these two factors will influence the development of a system.

It is ironic that “symbiosis between architecture and skills can change the character of the organization itself, as swamp guides become more valuable than architects” Some programmers see “Big Ball of Mud” as normal, and they even improve their skills on it. The cause is that architecture has been ignored for a long period of time. Later, this motivation to enhance “Big Ball of Mud” lasts. It is easier to establish a sophisticated architecture than a neat one. Skilled programmers tend to create a complex system until others cannot have good control of it. Maybe this situation is able to show their brilliance. This kind of codebase may become very hard to understand, and no one can totally comprehend it. “Once simple repairs become all day affairs, as the code turns to mud.” All programmers do every day is to fix issues that never stop emerging. Eventually, this system turns into a mud.

“Sometimes, complexity wins.” When a system becomes “Big Ball of Mud,” the complexity of this system is hard to imagine. Even though a skilled programmer wants to overcome it and fix issues, they sometimes fail due to the extreme complexity.

“Big Ball of Mud architectures often emerge from throw-away prototypes, or Throwaway Code, because the prototype is kept, or the disposable code is never disposed of.” It means that “Big Ball of Mud” is caused by “Throwaway Code.” In many cases, “Throwaway Code” is like an immediate solution or patch without careful verification. For resolving the emergent situation, these temporary solutions and patches are generated. However, no one proceeds the verification after applying. “Big Ball of Mud” of this system will gradually form.

Over the development of a system, “Piecemeal Growth” could emerge and influence

the system because people are encouraged to let the system work. Due to different growth rates of internal components, “Shearing Layers” emerge. This phenomenon gradually damages the architecture, and the consequence can be a “Big Ball of Mud.”

Before building a lasting and robust system, it is worth investing resources in exploration and experimentation in the prototype and expansion phases. The more people input, the more benefit they obtain. However, there are some activities that destroy the architecture, such as global variables or functions across the boundaries in consolidation phase.

“One of mud’s most effective enemies is sunshine.” Proceeding careful examination can help the code to be refactored, repaired and rehabilitated. Accordingly, “code review” is one of the useful approaches to make code clear and brief. “Pair programming” also can provide the similar effectiveness with “code review.” The more people touch and understand the code, the clearer the code they have.

During the “code review” and “pair programming,” these two methods also can share the experiences and wisdom with others. It is a natural approach to spread the information. On the contrary, if no one shares code with others, there is no motivation to make their code better because no one knows. In my previous software team, we held a group meeting weekly to discuss the commits in the recent week. We encouraged every team member to share their idea. I am convinced that this kind of meeting is a useful way to proceed group learning, and it is also very helpful for the development of projects.

Three approaches mentioned above to deal with “Big Balls of Mud.” “The first is to keep the system healthy.” This means that people need to maintain the system. By refactoring and repairing, they try to improve the quality of the architecture. “The second is to throw the system away and start over.” Sometimes this approach is the only option to do because the system is too complex to control, and no one can fix it. “The third is to simply surrender to entropy, and wallow in the mire.” Just give up repairing the system and let it go. Although it is not a good approach, people still adopt this approach due to some weird reasons. For example, some programmers prefer to stay in comfort zone and fix trivial bugs instead of doing something in a decisive and sweeping manner.

“You need an immediate fix for a small problem, or a quick prototype or proof of concept.” Sometimes, people generate a temporary prototype, and this prototype should be thrown away once people achieve their particular purposes. In fact, “Throwaway Code” still lasts in the system because this prototype is too successful to be replaced even though it is not well-designed. Usually, time is the major reason to keep “Throwaway Code” alive. As a result, this kind of code becomes reasonable, and then the system starts to be eroded. Programmers do not have enough time to generate an elegant code. Thus, “Throwaway Code” becomes an approach to fix the problem.

Prototype is a good way to understand the requirement, but it should not be considered as a final design. Consequently, the real issue is that “Throwaway Code” is still alive and is not thrown away. Two examples, PLoP online registration code and the Wiki-Wiki Web pages, are mentioned. Both systems might be considered that they are on the way from little balls of mud to “Big Balls of Mud.” Because these two systems needs to keep work, they are ongoing in a “Piecemeal” manner. If they can isolate the dirty code from other codes, it is a good way to prevent the whole system from being eroded.

The example of Russian Mir Space Station is shown that 1986’s scientists were not able to anticipate the development of ten years later despite such a precise field. Urban planning also reveals the fact that the master plan was not able to cover the whole city once places were far away with a certain distance from the center.

Before waterfall development, former programmers did not strictly define how to do “code-and-fix” in software development. Although it could be workable, this situation often resulted in “Big Ball of Mud.” Accordingly, waterfall development was created to fix this problem, but it didn’t deal well with complex software. “Larger projects demanded better planning and coordination.” Due to the complexity of larger projects, projects involved more people and more resources. There were many things needed to be considered.

“Problems were many times more expensive to fix during maintenance than during

design.” It is easy to explain in the current global world. One software is developed in one location and then released to everywhere. Once problems continue to emerge, the cost of on-site support should be much more expensive than generating a well-architecture at first.

Over time, new requirements always emerge, and customers need to have more flexibility and more choices. At the same time, the technology grows quickly. Consequently, these changes are usually out of our control. “Master plans are often rigid, misguided and out of data. User’s needs change time.” As a result, the architecture could not be locked down first, and it would be changed as time passed. Even though at first the graceful architecture exists, it will be eroded by the latter coming requirements and fixing patches. Every piece of software grows locally and individually, and then the software turns into “Big Ball of Mud.”

“Piecemeal Growth is based on the idea of repair.” In general, due to the lack of flexibility, programmers usually are forced to use patches which are global variables or cross-boundary functions to satisfy the requirements. Originally, no one wants to undermine the architecture, but the situation pushes people to do so.

“Piecemeal Growth can be undertaken in an opportunistic fashion.” Programmers can use every chance to re-examine and fine-tune the current architecture. Enhancing the whole software and preventing too much growth locally are necessary. Unorderly change can destroy the architecture. On the contrary, orderly can improve it.

“Extreme Programming also emphasizes testing as an integral part of the development process.” Testing should be involved at the beginning of projects. Programmers can get the feedback in the early phase and have a chance to adjust the system as early as possible. Therefore, feedback plays a significant role in software development.

“Probably the greatest factor that keeps us moving forward is that we use the system all the time.” In most cases, people need to keep their system working because of business requirements or living necessity. However, systems need maintenances. If the integration of maintenances is deferred, it will be hazardous. If the system is undermined, it is difficult to find which change impacts. Performing “Daily Build” at the end of every working day is a good approach to maintain the system. Programmers can get the feedback daily and repair the problems soon.

Although “Keep It Working” enhances “Piecemeal Growth,” refactoring can be applied. The system can work the same as before refactoring. In the meantime, unit and integration testing can assist this system to satisfy the customers’ requirements.

“Different artifacts change at different rates.” Every component could have different paces to change. These differences drive “Sharing Layers” to emerge. In software, software cannot keep unchanged. The general design idea of software is to put the components with the similar rates together. This kind of design can have high adaptability and stability, and this design seldom needs to re-design.

“The first step on the road to architectural integrity can be to identify the disordered parts of the system, and isolate them from the rest of it.” If the code is too difficult to understand and fix, the code often gets worse and erode other portions of the system. Therefore, separating it from other workable part is one of the viable solutions and then programmers can refactor it.

To deal with the mess of the code, the first step is to create a new interface to replace the old eroded one. This step needs skill, insight, and persistence, and it also costs time and money. Usually, programmers generate a new façade to substitute for the old one and do “Sweeping It Under The Rug.”

The example of Atlanta’s Fulton County Stadium shows us that it could not satisfy the current requirements, and then this stadium should be thrown away and start over. Obsolescence is the reason that pushes the original architecture to be replaced. Change is another factor to drive people to reconstruction because the new requirements do not include in the original design. People also consider the concern of cost in reconstruction because it takes time and money to perform. In the meantime, programmers need to get the support from their organization.

One reason to reconstruct could be that the people who created the system left.

Reconstruction could be a good approach to let novices have communication with the system. Another reason could be that programmers obtain experience from the previous system, and they could do better now.

As to “Reconstruction,” one way is incremental refactoring, programmers can abstract architectural elements and concepts from the previous system. An alternative way is reassessing whether new elements or structures can substitute for the whole of the partial system. If yes, programmers can save in reconstruction, repairing, and maintenance.