My Stint at Microsoft Development Org.

I was employed as a software development engineer at Microsoft Corporation between September 2012 and May 2014 located at Redmond. This was my first experience working full-time for any company (leaving out my prior internships) after I left graduate school. I am proud to have shipped Windows 2012 R2 (and Windows 8.1) on October 2013. This was a tremendous learning experience for me and an opportunity to see how a Fortune 100 company delivers enterprise scale software. In this document, I attempt to summarize my experience of dealing with the people and process at Microsoft development organization that I was part of. Anybody (and especially students since I was a college hire) who aspire to join Microsoft someday, or someone plain curious about the Microsoft culture, might find some interesting insights to learn from here. (Standard disclaimer: The content is based on my own opinion/interpretation only and does not reflect any opinion of Microsoft.)

My Background: Since my first interaction with a computer, I am a loyal user of Microsoft products. Back in 1994, when I first wrote a program for MS-DOS operating system in BASIC language I was fascinated at the range of possibilities that can be displayed with the black-and-white PCs (surprisingly, I never heard of the phrase "hello world" until I reached college in 2002). In those days, we used to load the operating system from a floppy disk. It was still an exhilarating experience for a ten year old kid. As time passed, I gradually grew up more comfortable in using Microsoft softwares. Fortunately in 2007 when I was about to graduate from college, I was offered a software developer job with Microsoft India Development Center located at Hyderabad, India. Despite the fabulous perks of working for the multi-national company, I decided to pursue higher studies. By that time I made up my mind to carry research work on Formal Verification due to the inspiration I had drawn from my undergraduate adviser and a couple of summer internships at University of Aachen, Germany. On June 27, 2007 I left my home country for Princeton University enrolled in the PhD program declining Microsoft's offer.

Traditionally, formal verification research was mostly focused on electronic design automation given the substantial funding from the largest silicon chip-maker company Intel. During my PhD, however, I chose to work on topics related to multi-threaded software formal verification. I made this deliberate switch to software partly due to the opportunities and promises that software presented during post-iPhone era. When I was in graduate school, smartphones started occupying people's pockets and everybody were writing apps for them. Cloud businesses of Amazon and Google were soaring. I thought to myself, I better be working on software related topics unless I want my research area to be irrelevant when I graduate. Around 2012, I realized that a by-product of 'mobile-first, cloud-first' (borrowing from Satya Nadella's mantra) era is large sets of data. And people who can infer interesting business insights from those data are going to be the rock-stars of the tech world. With little background in machine-learning or statistics, I knew I was not prepared for that kind of job (these roles in the industry are often referred to as data-scientists) right away. I did informational interviews with several people in the industry. In one of those info interviews, a researcher from IBM told me that I can probably slide into that role with either an internship or a relevant experience in a "cousin" technology. The option of internship was not on the table for me. Moreover, I thought Cloud based technologies are close ‘cousins’ and is a suitable candidate for making a transition. Although I interned at Intel and Synopsys (that writes software for companies like Intel), I knew that I shall be looking for jobs in cloud infrastructure based companies. At this point, I was offered a developer position in the CloudOS Infrastructure team within Windows Server division at Microsoft. I wanted to work for backend infrastructure teams where I thought I would get an opportunity to apply my knowledge of concurrency and distributed computing. (And boy, it had ‘cloud’ in its name!) So I accepted the offer.

What is an Org?: Among the few things that struck me during the first month at Microsoft was its strong chain of command within the development organizations or popularly known as “org” among the engineers. I have worked at other companies before but they were either entirely or partly research teams which worked mostly as a unit with much less visible external control. In Microsoft dev (At Microsoft, ‘dev’ interchangeably refers either to development or developer) org it was different. At Microsoft, engineers generally have one of three roles: Program Manager (PM) (one who answers the ‘why’ and ‘what’), Software Development Engineer (SDE) (one who answers the ‘how’) and Software Development Engineer in Test (SDET) (one who verifies if ‘how’ addresses ‘why’ and ‘what’). Although, we all worked together in teams, the chain of command was different for each role, e.g. in a team SDEs and SDETs report to different managers. In other words, at each level instead of one manager, Microsoft has three managers, one from each role (three heads are better than one in most cases anyway!). The rationale that I heard behind this was better management of resources (yes engineers are referred to as resources in MS!). This is true since often a PM can be working in more than one team simultaneously without having to report to multiple managers. These three chains merge at the level of Executive Vice President (EVP). Satya Nadella was our EVP before he became the CEO - I joined Microsoft as an SDE (or developer) in Windows Server and System Center.

What is Platform Software?: Microsoft started as an operating system company. In other words, it was in the business of building platform. In the software world, there are several layers. The lower layers interact either with the hardware or operating system. The upper layers run applications that users interact with. To give an example, the Minesweeper game is an application that runs on top of Windows operating system. The software layer that supports applications is generally referred to as a platform. In this case Windows operating system is a platform. At Microsoft, I joined the Windows Server division that builds such platform for running enterprise workloads. By enterprise, I meant big companies such as Delta, Intel, Macys etc. All companies need an IT infrastructure to run commercial applications (such as SAP - software that manages their inventory and personnel among other things). Traditionally, this infrastructure comprised of physical servers managed by IT administrators (or admins). However, due to cost and resource elasticity advantages, companies ‘rent’ virtual machines managed by providers like Amazon and Microsoft. In our org, we built infrastructure within Windows operating system to manage such large clusters of machine (either physical or virtual).

Building platform software is time consuming. It takes months if not years. For example, development of Vista took almost five years and development of Windows 8 took about three years. At the end of development work the software is ‘shipped’ or ‘released’ to customers. Many people might ask why does it take so much time to build platform software? Here’s why in my understanding. Microsoft followed a waterfall software engineering lifecycle model for a long long time. Usually, development of a software has four main phases: planning, development, testing and maintenance. Next I shall describe each phase as I have experienced it.

Phases of Building a Platform Software: During the planning phase, engineers brainstorm, survey customers, build prototypes for scenarios/products/features. At Microsoft, there is generally no dearth of new and cool ideas to work on. But the more important question that we ask ourselves is how much value are we going to create in the next release for our customers. The way we measure the value is by picking the right scenarios that we want to light up. One example of a scenario can be - "In the next version, I (the customer) should be able to watch a video in my phone from where I left it watching in my XBox console." As one can understand from this example, that this scenario crosses the boundary of products. We refer to this approach as Scenario Focused Engineering (SFE). Once we zero down on the scenarios, we prioritize them depending on their business values. Next, we break it down to sub-scenarios and then at the product/feature levels. The engineers are expected to estimate the cost of developing a new feature. As it is extremely difficult to estimate accurately, there is a notion called "T-shirt sizing", namely bucketising the cost into four broad groups, namely Extra-Large (XL), Large (L), Medium (M) and Small (S). Often engineers build prototypes to estimate the cost. Besides, we also do risk analysis, that is, what are the chances of failing to deliver within budgeted resources such as time and engineers. Risk associated with software development can be manifold. For example, working on critical legacy components such as kernel can be considered highly risky as many other components depend on it. Sometime a feature might need several other new features (built by multiple teams that rarely meet) to work properly, which can be also considered as a medium risk (more about it later). The testing can also be a challenge adding to the risk. If the ratio of value vs. risk is low the project usually gets deprioritized. This exploration of opportunities and choices, followed by conclusive decisions under consensus, takes several months. Generally, PMs are very busy during this time, often found running between successive meetings in the hallways. They also draft the functional specification of what needs to be built. This document discusses the scope, value and impact of the new feature. Next phase is design and development of the software.

At Microsoft, planning plays an important role everywhere since billions of dollars are always at stake! Before an engineer starts writing production code, (s)he needs to write the design specification. The developer lays out the technical details including, but not limited to, software architecture diagrams, contracts between components, source code layout, dependencies on existing software components and API additions/modifications. At the same time, test engineers start designing and writing the specification for testing the piece of software. Once the specs are ready, PM, devs and tests sign them off when all the open questions are answered. (Sometimes the sign-offs are conditional too!) Coding at Microsoft is very much regimented. There are sprints of 4-6 weeks dedicated to feature development those are known as coding milestones. Developers have tasks assigned against them such that each task can be done within 1-2 days. Once the code is written, there is a streamlined process for checking it in. It must be reviewed by a fellow developer/test engineer, known as code review. In my experience, code review is extremely necessary to ensure quality (such as style issues) and discover nasty bugs (such as memory leaks and code that breaks contracts) early enough. After code review, the change should pass the several tests including unit tests and functional tests. In fact, in my org the mandate was to write unit test for every check-in that covers at least 80% of the new code. Hence, unit test should always accompany a check-in that is additive. These coding milestones can be brutal if coding estimates are off. Developers try to leave some buffer for unforeseen events that I will describe next. In spite of all the good intentions, bugs get checked in. At the end of each day, the system attempts to build (or compile) the entire product. There are too many moving pieces in a complex beast such as an operating system and sometime bugs can lead to compilation/build failures in the entire source code branch blocking others. These incidents are referred to as 'build breaks'. Build breaks affect all since without a valid build (think of build as a timestamped snapshot of the product) the test automation framework won't work. And for the same reason, it creates lot of noise in the mailbox too. Somebody once jokingly told me, if you want to be (un)popular, just break the build. However, testing is not totally done yet.

Developers run tests to ensure no obvious bug gets checked in. But bugs can be of several complexities. The product/feature must also behave properly in end-to-end scenarios since developers make assumptions about the external components which may be incorrect. The product/feature should also pass stress tests since most platforms run forever. Stress tests ensure the software robustness under unconducive environments. Examples of such scenarios are network congestion or unavailability, too many memory intensive processes running simultaneously etc. Test engineers write code that can test feature code under all these possibilities. These tests are typically automatically run in the labs. Bugs are filed and tracked if any of them starts failing. The way the bugs are tracked at Microsoft worth mentioning. Testers file bug in a central bug database describing the steps to reproduce the bug (called 'repro steps'), and, the expected and the actual outcomes. These bugs are then triaged by the PM, dev and test leads. One should note that not all bugs are fixed right away. Most bugs during active development phase are fixed since they are mostly functional bugs. However, there can be performance bugs, stress bugs, spec bugs and many more. Depending on the cost/risk vs value of the bugs, leads decide either to fix them or resolve as "Won't Fix". And in certain rare cases, bugs can also be resolved as "By Design". (In my experience, "By design" resolution creates more back and forth between stakeholders than "Won't fix" since in the latter case the bug is at least acknowledged while in the former the legitimacy of the bug is denied.) The process of making this decision of fixing or not fixing is commonly referred to as 'bug triaging', typically participated by the PM, dev and test leads.

Risky Business: The entire process of shipping software has several risks of varying degrees. By risk, here I mean the odds of failing to deliver within allotted time. Therefore, development leads and managers try to minimize risks by employing various strategies. Scrums, weekly team meetings (or heart-beats), bug glides, bug-jail are several of those strategies. In daily/weekly scrums engineers report their status and discuss blocking issues. As bugs are filed against devs, they start gradually fixing them. Meanwhile, other bugs are discovered. Obviously, managers like to see that total number of bugs decrease over time. This decrease in bugs over time is often referred to as bug-glide. Eventually, as we arrive nearer to the shipping date, we hit a stage where bug count is zero and as new bugs are filed, they are immediately fixed. This stage is referred to as Zero Bug Bounce (ZBB). During active coding milestones (mentioned earlier), there is a concept called 'bug-jail' that enforces developers to maintain quality. A developer is typically allowed to accumulate at most n (=8) bugs while they are developing a feature. Once the count is more than n bugs, the developer must stop working on other things and get the count below eight. Until now I have described the process. Next, I shall share my insights that I have gained working here.

My Personal Insights: In my twenty months of experience working in a platform software team, I found the work challenging since any enterprise scale product should be functional, efficient, secure, robust and compliant with legacy components. (Trust me that is a lot of work!) In this section of the document, I will be discussing some of those challenges based on my personal experience, advice from seniors and hallway conversations. I refrain from suggesting solutions to them since there is no generic solution to those problems (that's why there are managers!). In practice, most of those problems are dealt with influence and experience of older folks.

In all the designs that I have worked on, I found that the pervasive theme was either one or a combination of the following steps: procure and filter data (by data I refer to the data that resides/generated in device/OS/other application), package them efficiently, transport them securely across machines/devices, store them persistently, and mechanisms to debug and configure any of the earlier steps. There was a significant application of concurrent execution involving thread-pools and locks. Hence obviously there is lot of concurrent state management. Given my background, I was curiously surprised, no formal technique is being employed to prove their correctness and ghost-bugs (or heisenbugs) haunt us sometimes six months after active development leaving devs with no clue how to repro them, let alone fix them. Apart from the development of this concurrent data-structures, most of the work does not require very smart algorithms. However, that does not mean that any part of it is less complex. Often complexity arises due to the constraints imposed by the transport protocols (e.g. unnecessary roundtrips between machines), software architecture (e.g. data wrapped under multiple layers and type system issues) and legacy components (spaghetti of binary dependencies). (One cannot just blame the engineers who designed those components a decade ago for failing to foresee today's requirements. Things like computational power, hard-disk space and network speed have drastically changed in the last decade.) Another source of frustration is building binaries for older platforms such as Windows Vista and Windows7 (there is a reason why Microsoft is retiring its support for WinXP). Due to business reasons, at least in our org, new products should also work as well in older platforms (or 'downlevels' as we call it). Typically the newer devs work on this thankless job commonly called Windows Targeted Release (WTR). We call it paying the WTR debt - everybody have to do it at least once!

The other challenge working within an infrastructure team is, designing a new feature on top of an existing infrastructure that is in maintenance mode. It is extremely difficult. Most often those legacy components were not designed to support today's scenarios and probably the dev who is currently owning it has no idea of 99% of its code-base (since its original devs have moved on many years ago). Sadly, these are generally the products with millions of customers depending on it daily. Sometimes a brave developer attempts to move a cog and all hell breaks loose, meaning random and seemingly unrelated tests start failing. Not so surprisingly, most new feature requests for these maintenance mode softwares are promptly turned down. I observed this dilemma in my org dev whether to build fresh new product (since that holds the most promises but has its own bag of unknowns and the burden of maintaining it once it has been shipped besides the existing ones) or to make incremental changes to existing products (that's where the current revenues are, but at the cost of missing out on new opportunities).

The value and productivity of a developer in a platform company grows with time. More than smartness, in my opinion, experience and knowledge of internals are more valued assets. A senior developer who knows exactly which function to look for in times of crises can save days, if not weeks. This happens primarily due to lack of documentation (and design specs are not enough to account for this). Often this knowledge is like unwritten tribal code that stays and disappears with group of people. Career-wise, I faced the cliched dilemma of depth vs. breadth. Should I continue to invest in learning more about the existing products and become a revered guru (or the go-to guy) in the org at the cost of losing touch with the world outside, or, do a bit more exploration before settling down? The decision rests entirely with individual engineers.

Dependency deadlock raises another form of risk. Suppose team A wants to depend on a feature that team B is owning. Under ideal conditions, team A likes to have the feature ready today. Unless the feature is available today, there is always a risk that team B might not deliver it before shipping because of variety of reasons (e.g. deprioritization). For team B, unless team A officially declares its intention, there is perhaps little to no motivation to invest engineer-hours into it. Therefore, team A cannot take dependency on team B since it is not ready and team B will not invest into it since there is not enough partner assurance from team A. Hence it turns out to be a classic chicken and egg problem as nobody calls the shotgun. The common sad outcome of such scenarios is that team A builds its own infrastructure that later competes with that of team B.

The future that I can see for this kind of orgs is, it will prioritize building end-to-end solutions/products more than just platforms. I agree that building a platform is super important. But nothing delights an enterprise customer more than an end-to-end scenario focused product. Moreover, building a working platform takes years, however, it is not necessarily useful without an appealing solution to a critical problem. This approach is more horizontal, where individual horizontal layers get built on top of another successively. The obvious drawback is that when the platform is released the customer needs have drifted probably. On the contrary, nailing one useful scenario across layers is useful right away. This approach is more vertical which targets end-to-end customer scenarios successively. Moreover, such solutions can be delivered at a much faster cadence and catering current business needs. At the company level, this shift in strategy seem to resonate well. This is because Microsoft currently is in the business of manufacturing phones, servers and playstation consoles besides shipping operating system.

In the book 'Lean In', author Sheryl Sandberg discusses her rule of 18 months in making career moves. She states that one should set short term career goals with about 18 months in the horizon. I feel she is spot on. Earlier Microsoft used to ship new versions of operating system once in 3-5 years. But technology horizon is changing so rapidly that this strategy doesn't make sense anymore. Since Windows 8 release in 2012, the company shifted into a faster cadence of release - about 1-1.5 year. I feel at an individual level, this works as well. One should have long-term as well as short-term goals that allow flexibility to change course and the agility to learn outside the comfort zone. A time-frame of 18 months, in my opinion, is perfect for an unbiased evaluation of those personal goals. Earlier in this document, I mentioned about the depth vs breadth dilemma. In my own career, I prefer to have more breadth at this point in time. (Historically, I find myself biased towards following a 'simulated annealing' based strategy[1] instead of a greedy approach, albeit circumstances permitting.) My long-term goal, at least now, is to work in the interface between research and development. Before, I have worked in the area of formal methods where we like to model all possible state transitions, if possible (analogous to the machine behavior). There is nothing fuzzy or uncertain about it. Not so surprisingly, systems engineering is mostly deterministic and finite. This is one end of the computer engineering spectrum. On the other hand, there is a growing demand in skill to deal with fuzzy, inaccurate and evolving set of data (closer to the human nature). This is the opposite end of the spectrum. In the short term, I want to acquire more skills in this end of the spectrum. In May 2014, I decided to move into a new role within Microsoft Research where I shall be expected to acquire skills in machine learning and search quality metrics. This is no doubt a risky move given my background. But I am sure, I won't regret the ride. And I am excited about it!

[1] http://en.wikipedia.org/wiki/Simulated_annealing: A strategy based on slow decrease in the probability of accepting (apparently) worse solutions as it explores the solution space.