Not so Agile?

by **Satyashri Mohanty**

In Greek Mythology, Procrustes, a bandit, killed his victims by force–fitting them into a standard iron bed. If the victims were shorter than the bed, he would forcefully stretch them; if taller, he would cut off their limbs as per the size of the bed.

This torture technique can be compared to how a predefined standard scale is used to forcefully "size up" work in an environment of high variability.

Sprint-based agile methodology of software development is suffering from the Procrustes Phenomenon. This concept's boundary conditions of applicability are highly limited in the real world. To understand why and how, one needs to go back to the history of software project management processes.

The Dubious History

The history of software projects delivery performance is dubious. Probabilistically speaking, as per the annual Chaos Report[1], software projects are most likely to fail in terms of cost, quality and timelines. Interestingly, the trend has remained the same through the years.

There is now wide spread consensus to declare Waterfall Methodology of software development as a major reason for the poor track record.

[1] *The CHAOS report by the Standish group is often cited as the de facto authority on success rates of IT projects. It was started in 1994. According to its 2015 report, 66% of technology projects (based on the analysis of 50,000 projects worldwide), end in partial or total failure. Interestingly, these statistics have been the same for the last five years" https://www.infoq.com/articles/standish-chaos-2015*

Problem with Waterfall Methodology: Process Over Speed?

The waterfall methodology meets two main objectives in software development projects

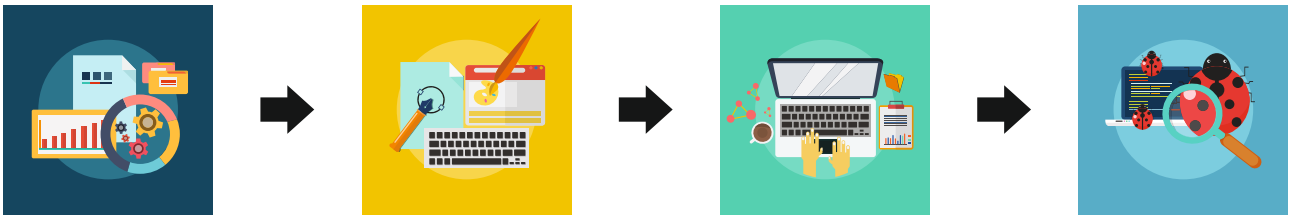
1 Ensure clear accountability in relationship between supplier (one who is developing the software) and the customer (one who will use it)
2 Ensure process is highly stable, and that level of rework is minimal

For the first need, freezing requirements before starting the project is an absolute must. Once requirements are frozen, the accountability for any effort or time overrun can be clearly traced back to either of the stakeholders.

The second need requires one to approach software development like one would development of a new physical product. Developing a new physical product requires the design to be complete before start of the actual building. Allowing the design to remain fluid during implementation leads to uncontrolled rework and interruptions.

Borrowing from these principles, a waterfall methodology requires one to move the project into following phases sequentially with clear "gates", marking completion of each phase

1. Requirements
2. Design
3. Development
4. Testing



Requirements Design Development Testing

Strict change management processes are to be put in place as a "deterrent" against any rethink late in the process. As a result, one of the overheads of this methodology is lot of documentation and sign-offs.

However, when the customer-testing phase comes at the fag end of the entire project, "hindsight" clarity issues create severe conflicts between customer and developer on what was "told" initially in the requirement phase and what was "understood". So, practically speaking, a strict change management process becomes difficult to implement, more so when the relationship between customer and supplier is that of unequal. Resultant effect is scope rework detected too late in the process, leading to uncontrolled delays. It is no surprise that contractual conflicts, delays, effort spiking and associated stress towards the end of the project are chronic to software development projects.

Agile Development: Speed over Process?

In sharp contrast, as the proponents claim, Agile Methodology takes a viewpoint that scope and design cannot be frozen upfront. Hence, it adopts a flexible working approach, which can embrace change much later in the development cycle. Instead of taking a "one perfect project delivery" viewpoint of software development, Agile believes in the principles of delivering usable features, good enough for customers

to start using, and then keep improving based on frequent feedback. The idea is to deliver workable software as soon as possible and get feedback for further iterative development. This way of working also takes away the need to freeze requirements and design upfront.

So, in the agile world, one can move away from sequential working between phases of software development. The team can work as a self-organizing group, which can design, develop, test and deliver workable features in an iterative manner. Time is managed by using concept of sprints (widely used version of Agile called Scrum) – a "time box" of 15/30 days, is used to deliver small batches of usable software features. While the ongoing sprint is frozen, subsequent ones can be changed based on customer feedback.

The "Not So" Silver Bullet

The silver bullet has been found. Or so, it seems. However, no conclusive, transparent and non-debatable study is available to establish the supremacy of Scrum Methodology[2].

Beyond the data, what is of concern about Scrum in specific are growing voices of practitioners against the methodology. Numerous blogs[3] have been highlighting the increasing stress of developers, the ever-lengthening list of bugs' backlog and customer dissatisfaction on usability of software. Surprisingly these effects are exactly opposite of what Agile had promised!

As a counter argument, proponents of the methodology have pointed out that implementation gaps, not conceptual gaps, are the primary reason for the woes.

The argument of implementation gaps puts the onus of the problem of poor results on the abilities of the implementer rather than the inventor. ("They couldn't implement – they were not disciplined!"). This narrative allows some management theories to stay as fads, longer than they really deserve to. So it is important to verbalize the hidden assumptions (or the boundary conditions) of a theory and check its validity in the practical world.

Flawed Assumptions

A widely cherished principle of agile is "self-organizing teams over strict processes". The signoffs/handovers/checks or any kind of formal control for sequential movement as per a process is scoffed at, as self-organizing and iterative development becomes the guiding mantra[4].

The resultant effect of any agile intervention is the re-organization of a functional structure of design, development and testing into many small self-organizing teams, which are expected to be self sufficient in terms of skills of design, development and testing.

This assumption of self-sufficiency of small divided teams in terms of adequacy of high-end skills, particularly designing and requirement assessment is highly questionable in vast majority of environments. High attrition rates, with many developers changing jobs every two to three years are hard realities of the industry, which is obsessed with outsourcing and reducing cost of development. The high attrition rate

[2] *"There is little scientific support for many of the claims made by the agile community"*
T. Dybå, T. Dingsøyr, "Empirical Studies of Agile Software Development: a Systematic Review", Journal of Information and Software Technology 50 (2008), 2008, pp. 833-859.

[3] *An Anti -Agile movement has been formed, its signature can be found on blog posts and comments across the net; there is even an anti-agile manifesto.https://web.archive.org/web/20160205025823/http://antiagilemanifesto.com/*

[4] *Agile Principles and Agile Manifesto*

does not impact availability of generic skill resources like expertise of a programming language. But environment specific skills like design and requirement analysis which is only acquired with adequate experience, are never available in full strength as required by the multiple decentralized agile teams. These skills (also called SMEs or subject matter experts) are always part of a miniscule minority in most environments.

So it is highly likely, with skills decentralization after an agile intervention, that many agile teams will fall short of skills, particularly those contributed by experienced SMEs. Even though, each team may have the required numbers to be declared “self-sufficient” as per records.

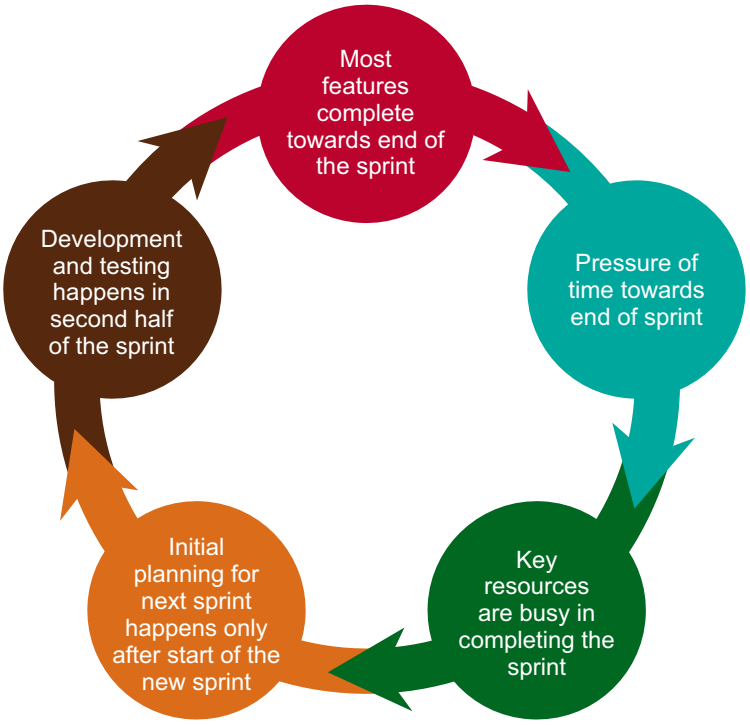
This leads to a situation where some agile teams depend informally on others, outside of their team, for help. With decentralization, some teams also become ignorant that they need help outside of their structure! Without a formal process to control sequential movement, the first casualty is adequacy of design and requirement assessment. Hence, testing becomes the primary source of feedback on design and development gaps.

Consequently, frequent re-work becomes inevitable. The lines of codes for same functionality are written many times over in an agile environment. At the same time, continuous “flow back” from testing causes frequent interruptions with developers. The problem of interruptions and rework further aggravates when different small agile groups are working independently on different sets of features in parallel; design decisions, at times, become “local” which cause conflicts between teams and create further rework.

The few SMEs or designers in some self-organizing teams become overloaded. These scarce resources are forced to multi-task beyond their capacity; they have to not just work within their team but also get into issue management of other teams. This further deteriorates the productivity of the system as everyone waits for feedback from heavily loaded resources. End result is an uncontrolled development process.

Agile has an antidote for this inherent chaotic working approach – sprint deadlines. Like Procrustes's standard iron bed, the sprint cycle deadline force-fits open work fronts into closures. Effort spiking close to sprint end pushes everyone to somehow drive closures and complete the sprint. But the efforts spiking compels compromises – some testing is skipped or some bugs are kept aside, some scope is set aside for later sprints – the same way Procrustes force-fitted human bodies into beds. As almost all resources in the team become extremely busy towards a sprint closure, any planning activity of subsequent sprint is put off to the start of the next sprint.

Consequently, in every sprint, initial time is used up in planning work. Actual development and testing happens towards second half of a sprint cycle, thus building pressure of time, followed by skips and misses. Testing resources stay less utilized during beginning of a sprint and then get overloaded at the end, where all planned features land up together. Sprint end rush and compromises becomes inevitable! The damaging effect of “forcing” a standard sprint cycle is not just seen in execution but also in planning. The arbitrary “cut” of deliverables based on a standard time, at times, lead sprint managers to size out work, which may not be usable at the end of the sprint.



When execution is chaotic, and features coming out from a sprint cycle are not "usable", the predictability for end customer is lost, particularly for those who might want a firm release schedule for a set of usable features to plan their own projects.

So, in many environments, it is not uncommon to find a hidden waterfall methodology-based project management imposed upon agile working. The resultant effect of this dual environment is a situation, which imbibes the worst of both solutions in terms of productivity losses.

The Hybrid Solution

Typically, an overall project plan is created for "final release" for customer commitment. The longer-term plan is broken into "sprints" with every sprint having its own predefined scope plan. The commitment to end customer is the final release. Sprints turn into typical milestones of an overall project schedule.

When many series of consequent sprints are pre-defined with scope, flexibility to re-scope or repeat a sprint based on feedback of previous sprints becomes nearly impossible. So after every sprint, the pressure is to work on scope of next sprint rather than to deal with feedback/issues of the previous one.

This creates an ever-increasing bug list and leads to massive effort spiking towards the end of the "final release". Some companies leave aside a time box of "final sprint" before final release to clear all the old mess of previous sprints. So in the

The agile principle of iterative process as opposed to sequential movement between phases has lead to bad practices of inadequate design and requirement assessment.

real sense, customers don't get anything usable till the final release.

At times, even the “buffer” of one sprint is not enough to make up for all the past sins. Less available time, pressure to deliver and overload of pending work is a perfect recipe for more compromises. Bad quality products are likely to be delivered to customers who report the bugs back to the development team. Pressure to expand the team to tackle the huge list of bugs of previous releases while developing for new releases increases the cost of development. Stress, poor productivity, delays and bad quality cannot be a rare phenomenon in such an environment.

At the same time, when an overall release project is super-imposed on agile cycles, the typical initial waste of time and development capacity spent in assessing the entire project efforts, scope and plan, negotiations becomes a reality. Software development companies lose out from all sides.

Agile Vs. Waterfall: Solution of Extremes?

One the assumptions of waterfall method, which was questioned by Agile, is the ability of customers to clearly specify requirements upfront. Many times, a requirement becomes clear only when customers actually see or start using something. At the same time, the need for perfection or the “best” can make the requirements phase go out of control. Hence, Agile did away with the overhead of trying to be perfect in design and requirements gathering.

But at same time, Agile implementations also suffer from bane of inadequate designs. What we have now is a more generic conflict underlying the software development process.

- On one hand, we should freeze designs and requirements before implementation because this approach ensures rework is under control and hence, time is under control.
- On the other hand, we should not freeze design and requirements because it is impossible to freeze them. Users get clarity on what they actually want after using or visually observing the product.

The only way out of the above conflict is to understand that there are clearly two types of rework which gets generated in any new product development process.

Type A (Foresight-led rework): They are misses and skips made under pressure of time and lack of synchronization between team members. Some people in the team know about these inputs. Due to pressure of time, these inputs are not incorporated in time; this leads to rework later in the process. These errors cause frustration among team members for missing the obvious. Using testing resources to get feedback on Type A errors is a criminal waste of time and capacity.

Type B (Hindsight-led rework): This is rework created due to new knowledge generated after an event of testing or clear visualization. It is in a way value adding, because new knowledge is generated in the process. Prototypes generate Type 2 value-adding rework. Type B rework is inevitable and no amount of thought experiment can help one visualize the Type B errors upfront. The only way out is to devise a process to generate Type B errors early in the development cycle.

Waterfall Method implicitly assumes that all errors are Type A and hence should be prevented. This school of thought has inspired the manufacturing slogan “First Time Right”. Process rigidity with checks and “no-go” gates becomes the key to avoid Type A lead errors. Agile assumes all errors are Type B, hence it considers one-time handovers between phases as waste of time. Hence agile propagates frequent back and forth between all phases.

Learning from History of Automobile Manufacturing

Interestingly, this "Agile" way is exactly how automobiles used to be manufactured in late 18th century. It was called the craft manufacturing method. A small group of mixed skill resource groups were each given a car to be assembled from start till end. The cross-functional team had skills of design, fitting and even machine operations. The not-so-perfect parts could be filed and adjusted to fit with each other, and eventually a vehicle would come out with arduous rework. This way of car manufacturing was indeed very expensive and only the rich could afford it in that era.

When Henry Ford presented the assembly line concept with clear division of labor, car assembly was transformed. Clear sequential working with different resources specializing in different types of the assembly task, defined the assembly lines of Ford Motors. The cars moved between stations (instead of the workers moving) for continuous flow of output. The specialization on one type of task, along with avoidance of switching losses (of craft manufacturing) between different types of skill work enabled a productivity jump of close to five times[5].

The revolutionary idea dramatically reduced the cost of making a car. The masses could afford a car because of the enhanced productivity. In a way, the car industry gave up the "agile methodology" of car making and achieved giant strides in productivity.

However, an assembly line working approach requires two critical conditions to be successful

1 Flow backs are negligible. Work moves forward in one direction. If nature of work is such that it frequently comes back to previous workstations for rework, then it can create chaos with uncontrolled line stoppages and reduced output. (In such an environment, craft manufacturing, or the agile method, is a better way). In case of Ford , assembly line working was enabled by standardization of parts, which could fit any car, without the need to file and adjust ill-fitting parts for a specific car, thus preventing the flow backs

2 Transfer batches should be small (ideally a single piece flow). Transferring of big batches not only expands lead-time in every stage but also leads to late detection of quality issues.

Rapid Feature Flow Model

Let us examine how we can get above two critical conditions to resolve the inherent conflict of software development process.

1. **Minimizing Flow Backs**

   Organizing resource groups into requirements assessment team, design team, development and testing group help create required division of labor for assembly line working. This allows the limited expert resources to be centralized in the requirements and the design groups. This grouping also helps in proper utilization of their capacity in type of work, which requires their expertise. However, without clear definition of what constitutes end of requirement and end of design, the assembly line type sequential working between resource groups will never materialize due to flow backs. It is also important to put up a process of check to mark completions – this is imperative to break the bad habit of proceeding without completing the phases.

[5] *In 1914, 13,000 workers at Ford made 260,720 cars. By comparison, in the rest of the industry, it took 66,350 workers to make 286,770 cars*

Requirements Phase

This phase should focus not only defining what should be done in a feature but also defining what will not be given. It is also important to begin with the end in mind. The inputs of all test cases to be used in the final validation should be defined. At times, visual prototypes can also help customers visualize what they really want.

Design Phase

A high-level dependency understanding at an architectural level is usually done in many environments by using the first few sprints, (called the foundational sprints). Dependencies which are neglected are in the subsequent functional sprints where features are delivered.

Design stage should focus on unearthing detailed dependencies in terms of interaction with different functionalities. Identifying verification test cases upfront in this stage makes sure that the gap related to any kind of dependencies which can lead to regression are minimized. In the absence of proper design, lot of capacity wastage can happen later in verification testing as fixing one issue might lead to breaking of other things, creating a vicious loop leading to unpredictability in flow.

The gates of design and requirements along with principle of "beginning with end in mind" should help in preventing type A errors and, unearthing type B errors early in the process.

(Interestingly, engineering of physical products focuses a lot on understanding dependencies as the cost of rework during product building phase is high. For example, if the design of beams needs to accommodate the load bearing capacity of a moving crane in a factory-building project, it is important such design dependencies are thrashed in the engineering phase. If the same dependencies are detected in execution for physical products, the huge physical waste of material becomes visible. However, in the case of software projects, wastages for bad design are never visible; a code going wrong can be re-written in the virtual world after it is detected in testing. No rejection of material or dismantling of build structure happens in the software development world. This inherent advantage of software development is also a bane. When testing is used for facilitating good design, the rework and loss of productivity is colossal, even if there is no loss of physical material).

2. **Minimizing the Transfer Batch**

Just having an assembly line with delivery coming out after a period of six months or a year is also undesirable. When different customers require different features at different points of time, having one delivery date for a large set of features is of no use to any one. It is unreasonable to expect the customer to wait till all other features are complete. So, moving individual features through the assembly line in order of priority (sequence in which features are introduced) ensures better flow and a short lead-time in delivering workable features that individual customers want.

The way to make a transfer batch of one feature is to institute a WIP (work in process) limit control in the different resource group types. The work in process control implies that a new feature will be introduced to the resource group only when one under progress is complete as per the criteria set for handovers. The rule of "one in when one is out" ensures a single piece transfer batch flow at a feature level. (Due care should be taken to define features in a way that they are fairly independent entities).

WIP control rule implies that tasks cannot be prescheduled or cut out using the "knife" of a time-boxed sprint. The rule of "one in when one is out" will ensure output of features in a staggered manner and that load on testing is leveled out. (Instead of wave of lumpy work for testing in a sprint-based scheduling.) The staggered flow prevents temporary bottlenecks in the system.

The WIP control also aids transparency of queues, which in turn helps in better capacity planning between resource groups. Skill group sizes can be adjusted time to time to manage any temporary elongation of queue. *(Change in complexity of features introduced in the system or loss of resources in some groups can create temporarily lengthen queues).*

The practice of forcing start of a task or its completion because of arrival of pre-defined schedule is in direct contradiction to a pull system based on WIP controls. Hence scheduling has to be given up as a planning tool.

The above principles will help in rapid flow of features from this system. The rapid delivery of individual deployable features also helps promote "a-version-at a-time" thinking approach in requirements phase to keep it under control. Customers (or marketers of the software) can keep improving the features by initiating work for the next versions.

The suggested model of working, however, has the following drawbacks

1. The rule of WIP control makes scheduling of tasks redundant. In the absence of a time schedule, predictability is lost.
2. It is in direct conflict with typical sales planning process of a product company that works on annual or bi-annual software release plans.

Improving Predictability

The above system exposes queues in the system and helps differentiate touch time (actual work time) from waiting queues. If the features' content can be standardized as multiples of a unit work package, then it is possible to model the expected arrival time to provide the desired predictability. When flow improves, with drastic reduction in interruptions, priority changes and rework, the system predictability goes up many folds.

But it is important to use the scale of time to provide continuous predictability and not use it as a tool to drive "commitment" in execution or use as a scheduling decision. It is a fallacy that schedules, and commitment around schedules, prevent time wastage. In reality, wastage of time is prevented by daily process of issue resolution and close handholding by flow supervisors in respective groups.

The decoupling of prediction and commitment can be best understood with the analogy of driving using Google Maps. Google maps offer expected time of arrival looking at queues on road. But it is only a predictive tool, and the driver cannot commit on the time claimed by Google. What he can do effectively is only reduce wastage in every moment of driving. In case of an overall manager of

a system, the focus of predictability improvement comes from reducing queues in the system, and managing issues interrupting flow on a daily basis.

In this model, a feature is introduced in the system without trying to cut it out based on some arbitrary sprint cycle definition. The feature size introduced in the system is based on what is deployable.

Dynamic Software Release Planning

The new model mandates that one move away from managing software development as a large monolith project with detailed plan aiming for a single release. Final releases can be done as frequently as one can manage based on the release overheads. Single features or a set of features can be bundled into releases as and when one wants. Instead of committing to a release plan upfront for a year, marketers of the software can decide on release strategy based on dynamics of the market. This provides necessary flexibility to react to competition strategy, and hence change the queue of features in the waiting list to deal with market dynamics.

To summarize, software environments have had a chronic difficulty of not being able to clearly differentiate two independent measuring scales required to manage projects effectively

1. Scale of time (to provide predictability of when something will be completed)
2. Scale of work to understand size

Both are different; they should not be mixed. Since defining a standard scale for work content has always been a challenge, using time (the sprint cycle duration) as the proxy scale to “knife out” work in planning and in execution is source of all problems in the scrum methodology. Hence, the Procrustes Phenomenon becomes unavoidable. The suggested Rapid Feature Flow model offers a way to differentiate the usage of the two scales to escape the wrath of Procrustes.

So what happened to Procrustes? His reign of terror ended with his death at the hands of Theseus, the king of Athens!