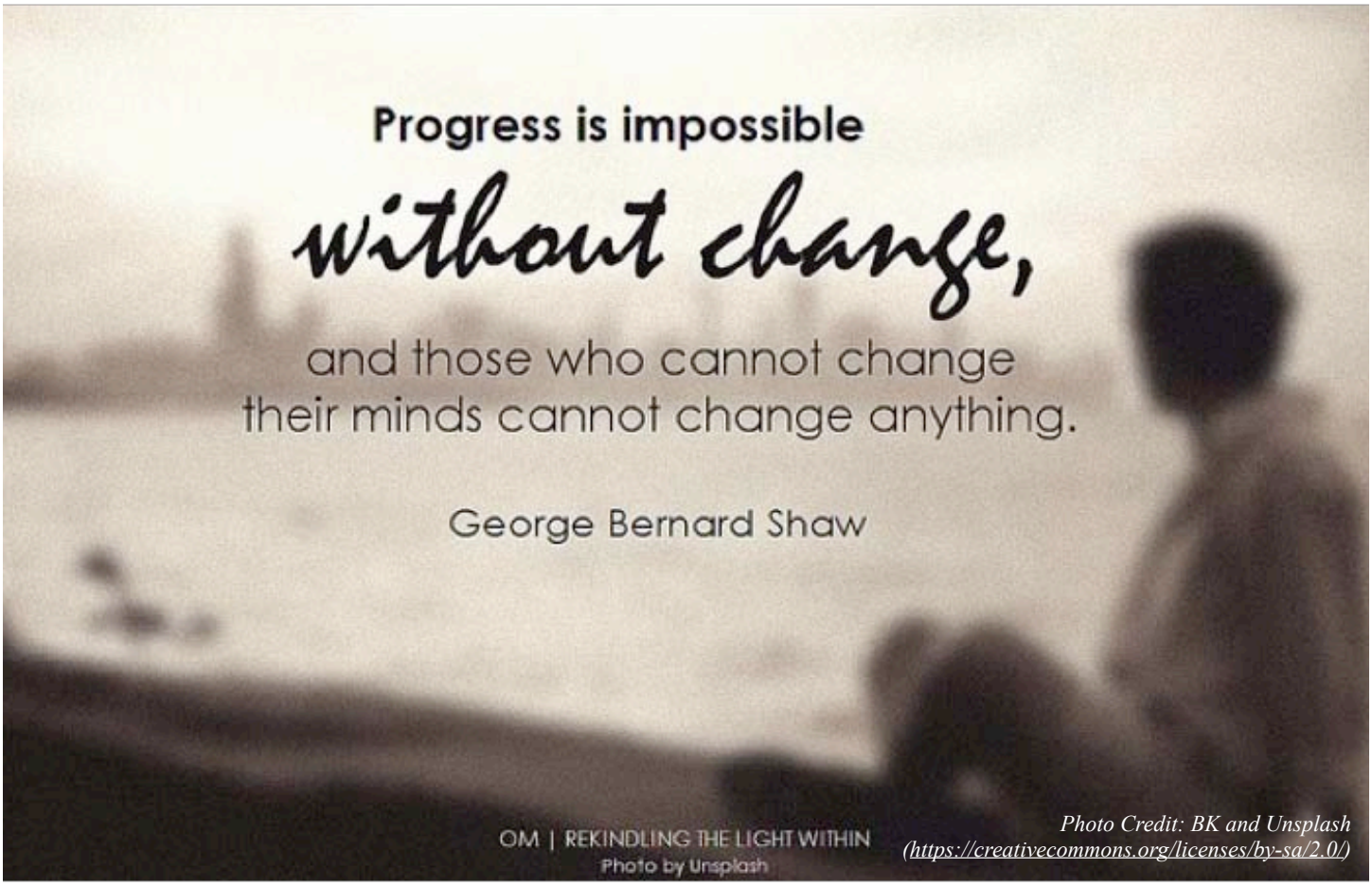


Agile Requirements:
Built to Change

By Bob Bretall
Enterprise Agile Coach

An often-quoted article on why the waterfall approach to requirements does not work is "IT Projects Sink or Swim" by M. Thomas from the British Computer Society Review, 2001. This famously states:

82% of projects cited incomplete and unstable requirements as the number one reason for failure.
The high ranking of changing business requirements suggests that any assumption that there will be little significant change to requirements once they have been documented is fundamentally flawed.
Scope management related to detailed up-front requirements [is] a significant contributing factor of failure.

"Failure" is kind of a loaded word here. Did these projects completely crash and burn? Some did. A portion of that 82% is projects that were canceled or never used. More commonly projects fail by either not delivering on time, exceeding budget (sometimes significantly), or both.

How do "incomplete and unstable" requirements lead to these failures? A couple of reasons that are pretty well known to most people who have been around software development for any length of time:

1. Attempting to specify all requirements at the beginning takes time and is not going to be fully accurate or complete anyway. Waiting for a fairly large requirements document to be completed delays start of work on development. It's true that they are usually doing other work while waiting, but that opens another can of worms with productivity lost to increased context switching and any delays getting the project underway have a tangible effect on the schedule.

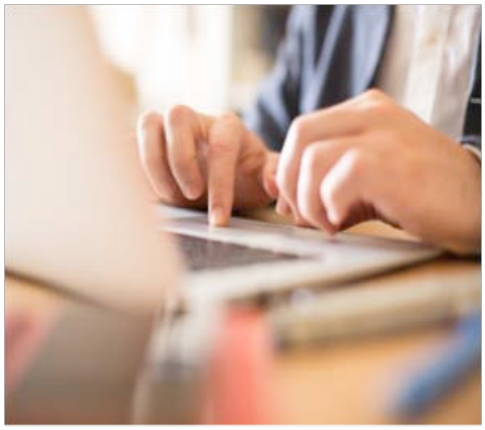
The high ranking of changing business requirements suggests that any assumption that there will be little significant change to requirements once they have been documented is fundamentally flawed.

2. Adding in and changing requirements as the project moves along without adjusting the budget and/or schedule to allow for this new work is very common. This is a lack of scope management, something I've always known as "scope creep". Some people think that the pointy-haired manager who always wants that one more little change is actually the "Scope Creep". It's fine to say "It's only one little change!", but when that "little change" is not planned for and is added to all the other "little changes" they can turn into quite a large unplanned effort and something has got to give. That "something" is usually the budget and/or schedule which are gleefully exceeded or disregarded, only to come back to haunt the people who have these kinds of things as a responsibility. The grief those managers get over missing their targets will generally flow downhill. Far better to control this at the source and prevent the problem before it becomes a problem.

Requirements will always change for any non-trivial development effort. As George Bernard Shaw said, "Progress is impossible without change". Locking down requirements is known to be unrealistic. Since we know requirements will change it would seem to make sense to embrace that change and have a process that not only acknowledges the changes by embraces them. An amateur marksman has an easier time hitting a stationary target than they do hitting a moving target. As their skill grows, they learn how to anticipate the motion of a moving target and aim at where the target is going to be instead of where it is at the moment they pull the trigger.

A successful software development effort is all about hitting the target. The requirements define the target. The changes we know are going to happen to the requirements as we go along add the motion. Anticipating the motion and building in methods to allow us to compensate for that motion and result in experts at software development getting just as good at hitting their moving target as expert marksmen are at hitting theirs. We need to anticipate where the target is going to be and make sure that is where we end up.

This brings us to Agile, which is all about building in a resilience to change. **Responding to change** is one of the core tenets of the Agile Manifesto. It is my opinion that being able to be responsive to change starts with the requirements. The requirements inform and shape our development all along the way. The requirements are there at the end letting us know we hit the target by passing the tests that show we meet the requirements. But wait! It's not quite as simple as that.



Responding to change is one of the core tenets of the Agile Manifesto. It is my opinion that being able to be responsive to change starts with the requirements. The requirements inform and shape our development all along the way.

Agile Requirements go hand-in-hand with Agile Development. There is no Agile development without Agile requirements. Without agility applied to your requirements and requirements integrated into the development effort throughout its lifecycle what we have is "code & fix" development. Code it up and cross your fingers while hoping you're developing code that is going to be what is ultimately needed. If it's not, you go back and fix it later. Remember, the requirements define the target. We want to hit the target which means keeping an eye on the requirements in every iteration and explicitly working to keep the evolving requirements in synch with development.

Following is a framework of notions that helped me codify my thinking about requirements:

- **It's not feasible to develop all requirements up front** – This is a concept that is generally accepted but we still have companies structured in such a way that they continue to put together complete (or nearly complete) requirements documents before handing them off to engineering even if they have moved to doing iterative development. Stop this. Requirements need to be iterative.
    - Customers don't necessarily know what they want up front
    - When they know what they want they can't necessarily describe it, but they'll know what they want when they see it
    - When they describe what they want, they will often jump to proposing a solution instead of focusing on what the real need is
- **Integrate requirements development into the development team** – this is a core tenet of iterative and incremental development
    - Acknowledge that the first cut at requirements is likely to have errors and incorrect assumptions
    - Embrace this uncertainty and continually check and refine requirements over time utilizing the feedback loop built into the very nature of iterative and incremental development as opposed to putting a large front-end loaded investment into creating low fidelity requirements that are far too detailed and will need to be redone over time anyway
- **Many types of requirements need to be considered -** Keep in mind that fully agile techniques of requirements definition (Product Backlog, user stories, etc.) are great for keeping requirements high level and not imposing needless solution-assumptions on the development team, but they are typically not adequate in and of themselves to handle system level concerns in larger product development efforts.

- Use the right requirements tools for the right parts of the problem (which means we will need to "scale up" as needed) – The team, program, and portfolio levels
- Even in Scrum (and other Agile methods) the Scrum-like team can NOT handle all the requirement necessary for anything but the smallest development efforts
    - Non-functional requirements are critical and can be missed when focusing on user-level features and story points.
    - **PURCS** – **P**erformance, **U**sability, **R**eliability, **C**ompliance, **S**calability
      (a slight tweak on the URPS paradigm I was taught almost 20 years ago in Dean Leffingwell's great book "Managing Software Requirements: A Unified Approach").
      Performance – A user is not going to wait 30 seconds for a response from the system or a screen transition.
      Usability – If your target customers cannot easily use the system, they won't use it at all!
      Reliability – Does the system crash a lot, if so, it's not going to be acceptable for use by the end user.
      Compliance – Does the software comply with all applicable standards? If you need to adhere to HIPPA standards and don't consider this from the ground up, you're dead in the water.
      Scalability – Does the system actually support the number of users required? What if we need to add another 1,000 or 10,000 or 100,000 users?

- **Requirements need to be considered at multiple levels in your organization -** Agile requirements are inextricably linked with transitioning to an Agile process
    - **Team Level** – User Stories flesh out the majority of functional requirements in the Backlog but Agile teams will sometimes miss the need for other requirements
        - **Other Work Items -** fixing defects from earlier sprints, support/maintenance activities, tooling & infrastructure work needed to support the development effort, etc. This is all work that needs doing and we need to account for it somewhere on the project schedule is going to get out of hand.
    - **Program Level** - Larger concerns here that transcend things that a small agile team will be fully aware of or capable of handling on their own
        - **Vision Document** – Address larger concerns like the strategic intent of the development effort, what problem it will solve, etc.
          The Vision that does not typically have time constraints, if we loaded the Vision with dates we would overpower WHAT is being created with too much concern of WHEN it will be created.

Agile Requirements go hand-in-hand with Agile Development. There is no Agile development without Agile requirements. Without agility applied to your requirements and requirements integrated into the development effort throughout its lifecycle what we have is "code & fix" development.

- **Roadmap** – A series of planned release dates that each has a theme and prioritized feature set (expands on the Vision, adding the time element)
- **Features** – Features are higher level expressions of requirements that must be decomposed into lower level requirements at the team level. (e.g. Provide the ability for users to rate the product using 1 to 5 stars)
- **Non-Functional Requirements** (see above) These start to heavily manifest at the Program level and serve as constraints on the backlog (e.g. when new things are developed we need to make sure to revisit NFRs and make sure the new stuff has not "broken" support for any pre-existing NFRs.
- Other things that needs to be considered at the Program level:
  - ***DevOps*** – Configuration Mgmt, automated builds & deployment, making sure the entire infrastructure is built and verified to support a deployable system.
  - ***System Level Test*** – Testing things in a full system context that is not necessarily feasible at the individual team level, but really an extension of the testing done at the team level.
  - ***System Level QA*** – Things that require specialty skills like performance test, load test, etc. Really making sure NFRs are met.

This only scratches the surface, but I hope it provides some good food for thought. There is nothing revolutionary here and it owes a huge debt to Dean Leffingwell's "Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise".
DESARA's Agile Requirements Workshop can provide help on implementing Agile requirements techniques in your organization.

THE DESARA GROUP

Bob Bretall

Enterprise Agile Coach

Bob Bretall has spent over 30 years in the software development field with hands-on experience in all aspects of the software development lifecycle. He has been a developer, designer, team lead, manager, trainer, mentor and consultant.

Read More **Bob.Bretall@DesaraGroup.com**