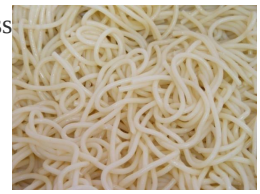
Managing Technical Dependencies

A common question asked in large enterprises when they start transitioning to Agile is ?What about dependencies?? When we talk about incremental development, evolutionary design and simple solutions some people get nervous. ?But our system is huge and complicated!? we hear. How can we apply Agile to something with a lot of cross-dependencies or external dependencies? Or even dependencies between Agile Teams? These questions come from our old-school habits of thinking that the more complex a system is, the more technical planning and design we need to do up front. And that we need to ensure that pieces get built in the right order to avoid dependency wait times.

Guess what? A lot of the complexity of enterprise systems came from doing Big Design Up Front (BDUF) and detailed technical planning. Not because they were designed to be complex (well, some were) but because BDUF precedes silo development which leads to integration surprises and ?stabilization? with late-project duct tape fixes and schedule pressure to cut quality resulting in overly complex systems that are maintenance headaches and enhancement nightmares. Or so I have heard. Oh, and have seen.

But I digress 

The topic was dependencies. We know from Agile Design principles that there are some ways to reduce the challenges of dependencies like making sure they run in only one direction, abstracting the functionality of a dependency behind an interface that hides the implementation and so forth. But still we have dependencies in new systems that leverage multiple technologies and we have legacy systems with tightly-coupled tangles of cross dependencies. Dependencies can give us heartburn because we may have to wait for a component and it may not do what we want it to do when it arrives.

Large-scale Agile development brings some new challenges to dependency management. These are nicely described in this great article from the people at Salesforce. Here is a summary.

- No one person or team can anticipate or know about all dependencies
- Teams have conflicting priorities across their individual backlogs
- Scope is dynamic so dependency management is ongoing
- With short release cycles, there is less time to coordinate
- Team completion dates may not align

So how do we handle dependencies in Agile product development when we don't want to spend a lot of time up front building a rigid plan that constrains our ability to respond to changes as we go? There are a few approaches that have emerged.

Types of Dependencies

A dependency is a service or component that the team needs in order to deliver functionality to their customer but cannot create for themselves. Here are a few classes of dependencies I can think of. You may know of some others.

- Third-party products or services
- Hardware that our software talks to, listens to or embeds
- Functionality described in a plan being worked on by a non-Agile group
- Functionality in the backlog of another Agile Team that we may need before they are interested in providing it
- A capability that is ours to provide but it appears to be associated with something having a lower priority in our Product

Backlog

A Depends on B

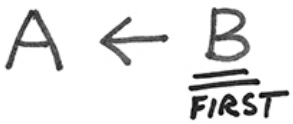
Approaches for Managing Dependencies

1. Feature Teams

An established practice in multi-team Agile product development is to form ?Feature Teams?, cross-functional teams that also have all of the skills needed to work vertically in the technology stack. Feature Teams are empowered to build what they need in order to implement whole User Stories and then merge and integrate using tools. Automated tests and continuous integration are essential for this to work.

2. Wait for It

Our instincts tell us that if A depends on B, we should do B first. Or, if we cannot do B ourselves then we will wait for B to be completed before doing A. This is the ?wait for it? approach, often a recipe for ulcers.



Instinct says to do B first

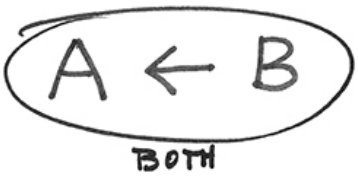
3. Test Double

Another approach is to do A first and assume that B will just work when it arrives. We can't ship without B but we can continue building our system and getting feedback by substituting a ?mock object? for B. I will describe some more success factors for this below. This is the ?test double? approach.

Do A first and assume B will work when it is available

4. Together

A Lean approach is to do A and B together if we can, whether by ourselves or in collaboration with another team. This approach reduces the risk of rework we have in the "wait for it" and "test double" approaches when A meets B and they turn out to be incompatible. By doing them together, each informs the other on what it needs and each will end up different than when they are done independently. This is the ?together? approach.



Do A and B together

Feature Team Approach

This approach applies to multi-Team product development. It complements the Togetherapproach to dependency management while providing many other benefits. It is well described in the literature. A good source is Larman and Vodde.

The ?Wait For It? Approach

This is our instictual approach: we cannot work on A until we have B. If you really can't affect the delivery date of the dependency, then you have to work around it in some way:

- Use an older version
- Build other features that do not depend on it
- Find an alternative

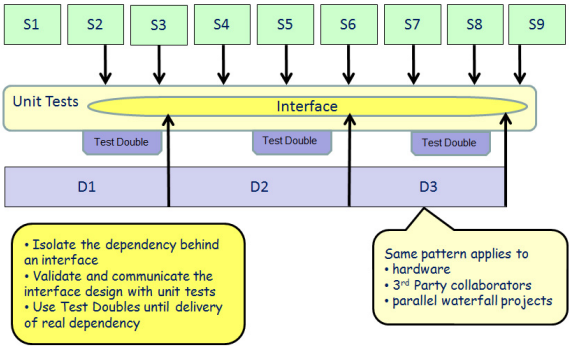
We always like to challenge our assumptions in Agile. Consider the possibility that we may actually be able to influence the delivery through various means including

- Bribes (beer is often effective)
- Bluffing (we need it so badly, we will do it ourselves)
- Borrowing (can we have one of your people on our team long enough to implement what we need?)

If the dependency is in our own Product Backlog, then it is simply a matter of changing the implementation order of items by realizing that dealing with the dependency sooner will reduce risk and increase value delivery even if the dependency appears to have a lower business value in itself.

The ?Test Double? Approach

Figure 5 is a complicated diagram but it illustrates a small number of concepts that help us succeed in doing A before B is ready. This diagram was inspired by the work of Jim Grenning though I cannot find the original article. He describes it as an approach for Agile development of embedded software but it can be used for many types of dependencies. When I present this in my Agile classes, I often have someone remark that they use this pattern and it works great.



Wrapping your dependency in tests

The concepts presented here are:

- Create a test double to represent the dependency until it is available. (For a discussion of test doubles see xunitpatterns. They are a superclass of the more familiar term ?mock object?.)]
- Abstract the dependency behind an interface, ideally one that hides the implementation.
- Wrap the interface in a suite of unit tests that clearly and unequivocally define your expectations for the behavior of the dependency.
- Share your unit tests with the dependency-builders. These are concrete, executable specifications that may speed up delivery and hopefully will enhance quality.
- Do not be surprised if the dependency-builders ask for changes to your requirements along the way. They will be learning from you.

The ?Together? Approach

This approach of doing A and B together works within the Team and between Agile teams, whether organized as feature teams or component teams. It is conceivable that you could combine it with one of the other approaches when dealing with a non-Agile provider if you can achieve a collaborative relationship.

One principle that Agile development inherits from the Lean tradition is Decentralized Control. It states simply that the people closest to the work are best suited to make decisions based on their intimate knowledge of the problem to be solved. To decisions, we can add ?agreements?. Rather than having dependencies managed by a Project Manager, Tech Lead, Architect or other individual who cannot help being a bottleneck in the flow of this work, Agile Teams are empowered to negotiate their own dependency agreements. This means that Teams may impact the prioritization of feature development for other teams, but enlightened Product Owners can learn to embrace this.

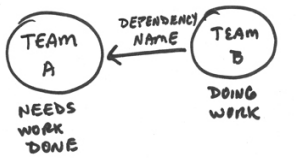
The ?together? approach is described nicely in the literature. One common manifestation is in the addition of a fourth question to the Scrum of Scrums meeting. The first three are familiar:

- What did my Team do since last meeting?
- What is my Team working on now?
- Is anything blocking my Team?

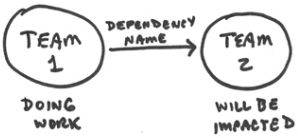
The fourth question is about dependencies-to-come:

- Is my Team about to do something that will block another Team?

Scrum of Scrums can itself be a bottleneck if it is the sole owner of dependency management. Work flows more smoothly when Teams manage the dependencies across their boundaries. A rich description of cross-team dependency management comes from the Salesforce article mentioned above. It describes a Release Planning exercise in which Teams identify dependent relationships between items in their backlogs and items in other Team backlogs and then negotiate agreements on delivery to minimize the risks. Two types of dependencies are identified: things we need and things we think other people will need. Check that article for more details of ongoing inter-team dependency management.



Traditional dependency flow



Proactive dependency flow

Conclusion

We have many options when dealing with dependencies beyond the traditional up-front planning approach. As with all things Agile they are based on self-organizing teams, collaboration, communication and priority. The next time someone asks you how we can possibly deal with dependencies when we switch to Agile, present these approaches to start dissolving the old-school way of thinking. You may experience a smoother development flow and end up with a higher quality, easier to enhance product as well.