Table of Contents

Why Should We Work Harder to Eliminate the Effect of Dependencies?

We all know that when you are trying to get a software product delivered, that dependencies are “bad”, but we also know that we will have them. I think, however, that we underestimate just how big an impact can have on the successful delivery of a project, with the implication that we should be doing a lot more eliminate, or at lease reduce, dependencies that are built into the way we deliver value.

How Bad is the Problem?

Risk Management

If you asked any project person “What is the ideal development situation” you would hear “the situation where I have total control of the situation.” In other words, a world where the work I am doing has no dependencies on any other work. Why is this the case? Its because I have total control over the success or failure of the project.

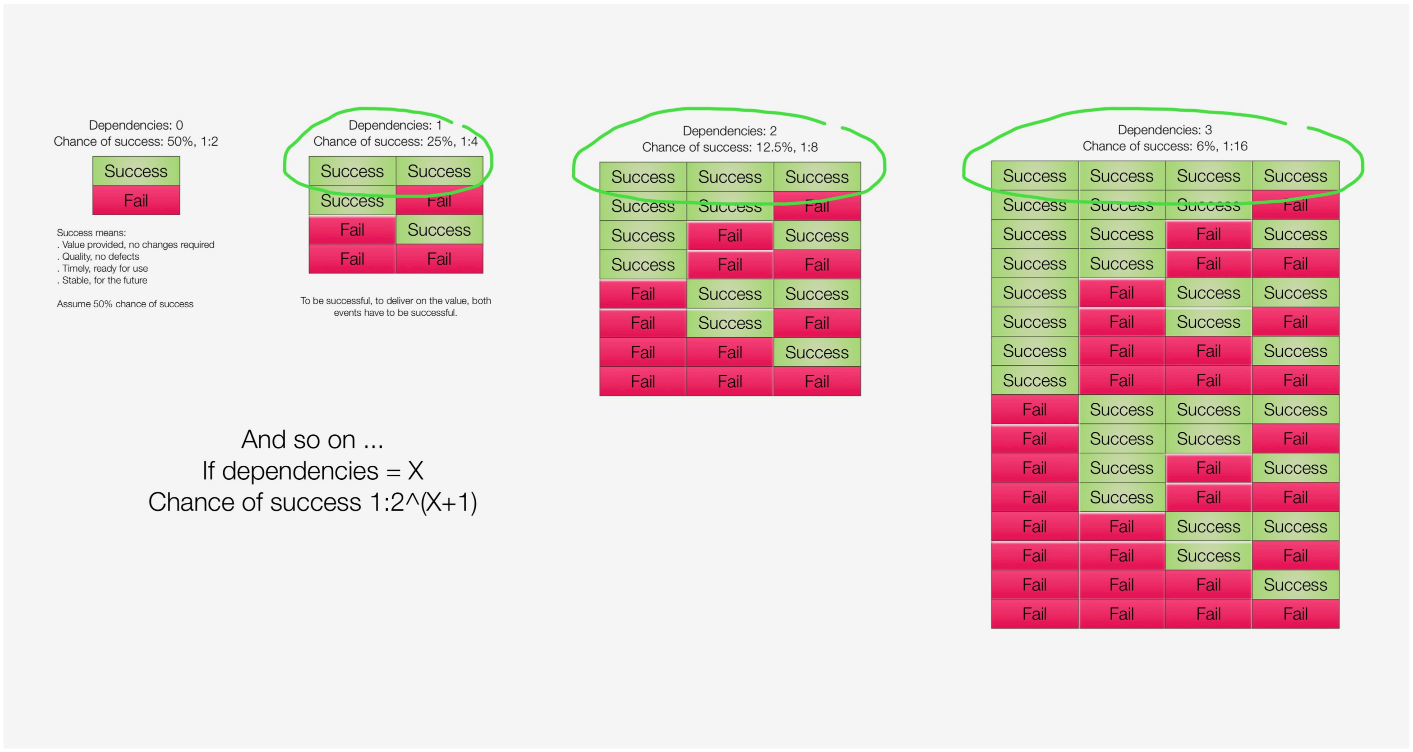
Lets say we are providing our project is to provide a service to another person. In other words, your customer is dependent on you, but you have no dependencies. You customer has a natural concern in that for her to be successful, not only does she have to be successful but you do too. Interestingly the success she has is not only dependent on you doing a good job, but also doing it on time so they can do their work. Success is “value provided”, “to quality standard / defect free”, and “timely”. If we do the work and deliver on time, but then get feedback that we have not got the answer we need, then from the perspective of the customer, we now have a failure to deliver the dependent capability.

Lets say our chance of delivering the dependent capability is 50% by this criteria; this sounds bad I know, but I always assume our customers will provide some feedback as a result of seeing what we have provided for them. For the customer of our capability to be successful you need to be successful, and the customer needs to be successful. There are 4 ways this can work out:

- Success / Success
- Success / Fail
- Fail / Success
- Fail / Fail

In other words, there is a 1 in 4 (or 25%) chance for your customer to be successful in the delivery of the capability and in the timely fashion.

But guys, this is just one dependency. What happens if we have 2, or 3 …


Dependencies: 0
Chance of success: 50%, 1:2
Success
Fail
Success means:
. Value provided, no changes required
. Quality, no defects
. Timely, ready for use
. Stable, for the future
Assume 50% chance of success
Dependencies: 1
Chance of success: 25%, 1:4
Success Success
Success Fail
Fail Success
Fail Fail
To be successful, to deliver on the value, both events have to be successful.
Dependencies: 2
Chance of success: 12.5%, 1:8
Success Success Success
Success Success Fail
Success Fail Success
Success Fail Fail
Fail Success Success
Fail Success Fail
Fail Fail Success
Fail Fail Fail
Dependencies: 3
Chance of success: 6%, 1:16
Success Success Success Success
Success Success Success Fail
Success Success Fail Success
Success Success Fail Fail
Success Fail Success Success
Success Fail Success Fail
Success Fail Fail Success
Success Fail Fail Fail
Fail Success Success Success
Fail Success Success Fail
Fail Success Fail Success
Fail Success Fail Fail
Fail Fail Success Success
Fail Fail Success Fail
Fail Fail Fail Success
Fail Fail Fail Fail
And so on ...
If dependencies = X
Chance of success 1:2^(X+1)

It turns out the chances of success are exponentially lower the more dependencies you have! For X dependencies, the chance of success is 1:2^(X+1) or as a table:

Dependencies (50%)	1 in X Chance	Percentage Chance
1	1:2	50.0%
2	1:4	25.0%
3	1:16	12.5%
4	1:32	6.3%
etc.	etc.	etc.

“But”, I hear you say, “we are much better than that 50%, even with that criteria. We operate typically at 80% success rate.” OK, I understand, although I suspect that is not true, but if we run the numbers that way, we get:

Dependencies (80%)	Percentage Chance
1	64%
2	51%
3	40%
4	33%

As you can see, even with that high success rate, a string of dependencies will quickly create a problem. With just 2 dependencies you are already a coin toss (about 50%) as to whether the whole thing will be delivered successfully.

Time To Market

> Dependencies means delays

There is more to this problem than a simple problem of risk management. One of the reasons people want to deploy an agile approach is that they want to reduce “time to market”. From a Feature perspective, we want to want to understand how long a Feature is “in progress” or cycle time. The problem with dependencies is that the more dependencies you have to release a Feature, the longer the time it will take to build.

How long will it take to deliver a Feature? If we assume that we have a common Iteration (Sprint) length and that in the normal case a team takes on and completes their work in a (Iteration) Sprint then a simple model for calculating the amount of time to deliver a Feature is (assuming you have a common Iteration (Sprint) cadence (i.e. all teams finish the Iteration (Sprint) on the same day):

> Cycle time for Feature with dependencies = sum (Iteration (Sprint) length for all teams in the dependency tree)

or, if you don't have a Iteration (Sprint) cadence:

> Cycle time for story with dependencies = sum (sprint length for all teams in the dependency tree) * 1.5

We can represent this as:

Dependencies	Number of Synchronized Iteration (Sprint)	Number of Unsynchronized Iteration (Sprint)
1	2.0	3.0
2	3.0	4.5
3	4.0	6.0
…	…	…
7	8.0	12.0

And so on.

What does this mean? If you are thinking “we want quartertly releases”, and your Features “require” work from a number of Teams (in other words, you are organized as a series of Teams that do not deliver whole Features but rather architectural layers of a product) then you have a problem.

You might argue that this thinking is too pessimistic. You will often see teams start and complete work mid-Iteration (Sprint) or even jointly work on items in parallel. Others argue that this thinking is too optimistic as teams produce bugs which introduce loops and additional delays in getting an item finished. In reality, the time to market is probably a lot worse than this in that we know that dependencies also lower the chance of delivering. In general I've found that while your specific numbers might vary this type of analysis is valid and can be easily made more realistic based on your data.

Loss of Knowledge

Simply …

> “Why is hand-off so destructive? Because the best teachers on their best days get across less than 30% of their knowledge. So, hand-off implies throwing away at least 70% of knowledge on its way to the people doing the work“ — Allen Ward

Every time you hand something over from one person or group to another, you lose knowledge, unless you spend a lot of time making sure the information needed also flows through.

Cognitive Load

The book "Team Topologies" by Matthew Skelton et al argues that in order to improve the flow of value you should organize Teams and Trains so they have low cognitive load.

There are three types of cognitive load:

- Intrinsic: What you need to know to get stuff done; aspects fundamental to the problem space
- Extraneous: Extra information needed to get the job done. For example knowledge about how to set up a development pipeline might be considered extraneous for large organizations if everyone is required to do have this knowledge.
- Germane: Additional context for high performance (e.g. API usage); this allows more cognitive space to add value.

In many cases cognitive load increases as a function of the number and type of dependencies we have, which again impacts the flow of value.

My Dependencies

Now my question to you - “how many dependencies do you have in your project and how deep are they”?

The “deep” question is based on thinking through “in order to deliver value of a specific item to a customer, how many layers do I have to touch?”

One of the problems with traditional software development has been that we increasingly created silos of knowledge based on specialization of disciplines. As a simplistic example, we'd have a team that would be responsible for the user interface, another for the business logic, still another for the data access. In order to deliver any functionality that the customer would regard as valuable, we immediately have created an environment where the probability of success is around 12% for any feature as there are 2 dependencies (assuming UX is the “driver”). Worse, all our features require a path through all these layers, and so we set up this “failure” environment for all features in a project.

Dependencies lurk everywhere:

1. All the features you are dependent on to get something to work
2. The skills you need to have access to when you don't have skills needed
3. Hardware and software environments to, for example, test and deploy software
4. Serial processes you have in place for governance reasons
5. And so on and so on and so on …

The more dependencies you have, the more likely it is that your project won't be successful.

What Can We Do?

The steps are simple to describe but, of course, hard to do for any decently sized project:

1. We need to understand where dependencies are and visualize them.
    1. Set it up so you track dependencies, whether that is through tagging on Kanban boards, a separate spreadsheet, whatever. To improve data collection, use a taxonomy (categories) of dependency such as knowledge, task, and resource dependencies (for more information on this see “A Taxonomy of Dependencies in Agile Software Development” by Diane Strode and Sid Huff). This approach can help pinpoint problems areas.
    2. One team I worked with pulled together a meeting of key technical people in a room and simply asked themselves “for a typically feature, what components do we have to touch and which people are responsible for these components?” A simple diagram then showed them how the software they had involve 7 deep levels of dependencies for a number of situations. This is a good starting point. Note: Jeff Sutherland calls this "organization debt".
    3. The SAFe folks talk about the establishment of a "program board" where, in a multi-team situation, a list of dependencies is maintained based on the work, iteration by iteration. This could be used during an Inspect and Adapt to ask the question “are we really set up to optimize delivery of value with the current dependencies we have in place?”
2. Our first approach should be to eliminate the dependencies. Some examples of ideas to address:
    1. The formation of an agile team oriented toward the value to be delivered, with all that is required is a good step.
    2. We can combine a number of components together so that instead of having layers, we create a single place to go. With the understanding of the “dependency issue” there are probably design changes that can be made to the product that will improve the chances of success.
    3. We can eliminate a need for an approval process by changing how we collaborate to develop the material. For example, perhaps the governance requires an “approved design”. Instead of going through the standard document / approve cycle, we could build the required documentation incrementally as we do the work, and then automatically approve when the work is completed.
    4. Perhaps we are dependent on another team for some work. Instead of keeping the dependency, we could take on the work for that team, completing both sets of work at the

    same time (note: I understand there potentially a lot to make this really work, but its probably worth the effort).
    5. We can invest in automation of dependencies which impact our cognitive load, especially extraneous cognitive load. Using the example above, we'd invest in automation of the development pipeline.
    6. And so on …
3. Our second approach should be to reduce the effect of dependencies. Some examples of ideas to address:
    1. The last example talked about a team taking on the work of the dependency. An alternative approach could be, to ensure that we produce what is wanted first time, to work in parallel, rather than in serial fashion. In other words, the two teams working to fulfill their part of the requirement at the same time. The way to think about this is through the idea of a "trust model":
        1. If you don't trust the people delivering on the dependency, then don't commit to dependent work in the sprint until the 3rd party delivers their work.
        2. If you do trust them, then the agree to work together during the Sprint to get the work complete.
        3. Work hard to establish the trust model as this will allow faster delivery and feedback cycles.
        4. Don't be silly about trusting - if you have the trust model and the 3rd party breaks the trust, then assume the "don't trust" model. Even if they are trusted, you might want to think "trust but verify."
    2. A slight variation on this is to determine if there is a way to slice the stories the two teams are working on into smaller slices, perhaps working on high risk areas first. Then team 1 completed the story early in the sprint, hands it over to team 2 to work the remainder. This way, when the next set of work comes around we are confident of delivery. And who knows, perhaps it can all be done in this sprint.
    3. Ensure that you have clear, unambiguous agreements (APIs, service level agreements, dates, etc.) between yourself and whoever you are dependent on.
    4. We could invest in an API strategy so that people don't need to understand everything about everything, but rather can just focus on using an API, dynamically reducing their germane cognitive load.
    5. Some governance procedures were put in place to address a specific issue but there are many times that no-one can remember why it is put in place, nor understand why it was important. In other words, there is no value in doing the work, but we still have the procedure in place ("organizational cruft"). Perhaps a review will result in fewer governance requirements, or changes in the way we meet governance issues, this reducing a slew of dependencies.
    6. And so on …
4. Our third approach is to ensure we have a backup plan in the instance that whatever we thought was going to happen didn't actually work out.

For those that follow approaches adopted by Amazon, they take an aggressive approach to dependencies. Basically the idea is that it is up to everyone to aggressively work dependencies in Amazon. It is seen as a sign of leadership. Jeff Bezos broke the approach down to 3 steps:

> 1. “Whenever possible, take over the dependencies so you don’t have to rely on someone else.
> 2. If that is impossible, negotiate and manage unambiguous and clear commitments from others.
> 3. Create hedges wherever possible. For every dependency, devise a fallback plan—a redundancy in a supply chain, for example.”

Note: You need to make sure that you really do work the dependency and improve your execution. For example just moving the work from 2 teams doing the work to 1 does not eliminate the dependency. It only potentially helps eliminate the dependency. If the team that takes on both sets of work does not materially change how they are going to compete the total work then you can expect only limited benefit from putting it in to one team. But I also think there is a chance for improved success because that option is available to the team and, at a minimum, learning from doing package 1 is available to the team which is also working the next package.

Want to Know More?

- “A Taxonomy of Dependencies in Agile Software Development” by Diane Strode and Sid Huff
- Impact of Multiple Team Dependencies in Software Development is a simulation put together by Troy Magennis where you can play with the effect of dependencies.

Enterprise, Scale, Dependencies, Risk, RiskManagement, FAQ, PresentationIdea, BlogEntry