Project Management guide

Project Management Triangle

Iron Cross of Project Management

Also found as Iron Cross of project management. The purpose of the adjective “Iron” in front of the “cross” is that you cant change one aspectof it without affecting the other. Iron Cross of Projcect management defines 4 software development constraints: Good Product (quality), Quickly delivered (time to market), Cheeply Delivered (cost under budget), Done (all requested features are delivered). Often balance is required: good enough, fast enough, cheep enough, done as much as possible, in descending order of priority (to adjust scope see estimating, prioritizing and buffering). Constraining all four simultaneously (or usually the scope, but the other just follow the scope nontheless) gets us the sequential project management - waterfall.

The Project Management Triangle toppled over

So the Project Management triangle is a similar visualisation of the cross but as a triangle - Schedule (time), Budget (cost), Scope (features); central area (quality). Sometimes it is better depicted as a pyramid. In agile we turn the triangle around. We “constrain” the time and resources, but we adjust the scope.

If we see the Scrum framework we have “Sprints” that are constraints of time and resources, i.e. duration is fixed to 1, 2 or 4 weeks, and the team size and membership is stable during that duration; however the scope can vary according to the velocity of the team, their forecast and the vision of the product owner for what brings highest value vs. what is “ready” to be developed.

There is a third dimension of the triangle, turning it into a pyramid. This is the quality aspect. So, if we have no other option but to constrain the time, scope and resources, that the quality will always suffer. To let the team be self managing, autonomous and with that responsible, we must let them be proud with what they build.

Some approaches consider other constraints as well, such as people and resources.

Protecting the constraints

Resources must be protected from uncertainty. Uncertainty manifests itself when the unplanned happens. A system can absorb uncertainty with the provision of buffers. In every case, a constraint can be protected by a buffer. A buffer would normally be allocated in the same unit of measure as the constraint is measured. Hence, people should be buffered with people, schedule with time, budget with money, functionality with requirements, and other resources with similar resources.

Protecting the People Constraint

As the single most important resource on a software project, people must be protected. There is more than one way to provide a buffer for the people. The most obvious is to have more developers than needed. The second is to assume that the people are only productive for a subset of the work day ~5.5 hours. This is a personal choice. Others may suggest different numbers. Buffers must also be introduced to accommodate vacations and ill-health. A wise manager will buffer a little more for unpredictable, life changing events such as a death, birth, marriage, terminal illness, and divorce and more esoteric ones.

How big a people buffer is needed? The answer is that buffer size varies with uncertainty. The greater the uncertainty, the larger the buffer must be. It is important that this people buffer is added to the project early. Brooks suggests that adding people later in a project is not effective; they must be added early.

Brooks' Law: Adding people to a late project makes it later.

Uncertainty will vary with the length of the project. With a very short project, there is little uncertainty about people. With a longer project, there is more scope for things to happen unexpectedly. However, it is better to think of software development as an on-going process where projects are simply inventory passing through a system of software production. In which case, it is reasonable to suggest that people-related uncertainty tends to level out at a predictable level.

Protecting the Time Constraint

Every finished product or project has a desired delivery date or a predicted delivery date. A desired delivery date is when the customer asked for delivery. A predicted date is when Engineering (or IT) estimated they could deliver it. The important thing is that a date is agreed upon by the software development organization and the customer. This date is a system constraint. A date gets promised, and it is bad business to break promises.

If the delivery date is a constraint, it must be protected by a time buffer. The length of the time buffer must be based on the uncertainty involved in the project. The uncertainty must be assessed with respect to the technology, people, subject matter, architecture and delivery environment, the maturity of the organization, and a number of other factors such as the reliability of upstream suppliers.

PICTURE

Certainty is the percentage likelihood that a task will be completed on time. The 100% certainty implies that similar tasks always complete on time. If only one in every two tasks tends to complete on time, then the certainty would be 50%. In reality, certainty is the gut feeling of the developers who are on the spot. If they feel confident, they may estimate 90% certainty. If they've never seen something before and have no idea how to do it, they are more likely to estimate <50% certainty. For example, if a developer estimates a job as 1 month, with a 40% certainty of completion, then the estimate should ideally be inflated by 2 to 3 months.

Protecting the Scope Constraint

The functionality of the system represents another constraint. Once again, the agreed scope represents a promise. Failing to deliver the agreed functionality means a broken promise. Broken promises are bad for business. If the scope constraint is to be protected, it must be buffered with more scope. In essence, the manager must gain agreement with the customer that some of the scope would be "nice to have" but doesn't need to make it into the currently agreed deliverable.

In other words, if the scope constraint is to be protected, then the scope of required functionality must be prioritized. The customer (or product marketing) must determine the ranked order of importance for the functionality in the scope. This could be as simple as assigning a value on a 3-point scale to each required function, for example, "must have," "preferred," and "nice to have." Functionality priority is actually a two-dimensional problem. When a customer is asked how important any given feature in the scope might be, the answer is likely to be, "It depends." What the customer means is that the importance varies according to the delivery date.

The program manager should plan to agree on a scope with the customer that includes a number of "nice to have" requirements that perhaps later become "must have." Engineering would agree to schedule these requirements in the plan, but the customer agrees to let them slip, in the event of unplanned events disrupting the schedule.

As a precursor to ever being able to buffer scope, a development organization must provide end-to-end traceability, transparency, and repeatability. It must be capable of building trust with its customers and showing that it can deliver on promises. Only when the customer learns to trust the software development organization, through an understanding gained from visibility into the software process, will the customer

agree to prioritize and buffer scope.
Hence, it should be taken de facto that scope buffering is not available to the immature software production system as a protection mechanism against uncertainty. Trust must be earned through a series of deliveries that met expectations. After this, it is possible to negotiate a scope buffer.

Protecting the Budget Constraint

The budget for a development project is very important. If it is underestimated, money may run out and the project will fail to deliver. Failure to have enough money may result in a broken promise. Exceeding the budget represents another broken promise. Broken promises are bad for business.
Budget is buffered with money. It is that simple. How much money is needed? Once again, the size of the buffer will depend on the uncertainty. General operating expenses have a low uncertainty. Generally, the cost of the developers and their immediate overheads for office space and other costs are relatively predictable. A buffer will be needed to cope with technology and subject matter uncertainty. In other words, the larger the time buffer, the larger the budget buffer must be.
A budget buffer is needed to cope with time overrun, that is, use of the time buffer. More may be needed to cope with unforeseen costs related to technology or subject matter. When the subject matter for a new software product is poorly understood, the estimate of its complexity may be in error. As the scope is investigated during analysis and design, an area of the requirements may prove to be much more involved than previously thought. This may result in the estimate of the inventory in the system increasing.

Protecting the Resource Constraints

It is important to remember that because software is an innately human activity, the majority of the costs are incurred by the people. If a day is lost from a developer’s time because a PC broke down, it is possible that Throughput for the whole system may have been lost. It is, therefore, vital to know where the constraint is within the system of software production. If a developer works in the constraint and produces a unit of production per day and the software production system averages ¤10,000 of Throughput per unit, then the cost of losing one developer for only one day could be ¤10,000.

Summary

Uncertainty in software production is inevitable. There are five general constraints in software development—delivery date, budget, people, scope, and resources.

Uncertainty can apply to any of the constraints. Uncertainty is addressed through provision of a buffer. The buffer is generally provided in the same form as the constraint, for example, people are buffered with more people.
The provision of buffers on constraints is easier to negotiate under conditions of trust between the supplier and the customer. Trust is built over time with delivery of agreed functionality. Trust is maintained through transparency of process and mutual understanding.
There are four types of uncertainty—variation, foreseen, unforeseen, and chaos. The types of uncertainty encountered in a given software project will affect the size of the required buffers.

Vacation PBI vs Capacity calculation (how to plan for velocity deviations)

Lets say that we have tracked the velocity of the team for the last 5 sprints and we have:

- Average velocity of the team, hence their “yesterday’s weather” forecast for the next sprint.
- Deviation in the average velocity, or high and low forecast, hopefully getting lower following the “cone of uncertainty” rule.
- Average velocity per team member.

It is only natural that a Project Manager will now try to calculate the capacity of the team for the next iteration. Let’s say that we will have Jane off for two days, and we will have Joe for 8. We have the average velocity of Jane and Joe, or if we want to be more fair, since we have calculated the average per team member (or per seniority ranking of team members) we will use that and reduce the velocity by the calculated amount. This of course is not a good agile approach since we are reducing story points back to ideal man-days, since the story points are linked to the size and effort of a story or issue at hand and not to its people. Although this is how Issue Tracking tools basicly work so we need to consider this approach as well.So we end up with, lets say capacity = 0.75 * avg.velocity.

Now, lets try a different approach. Let’s say that we come to the Sprint Planning session and we say, “ok team, we have some vacations this sprint, Jane and Joe will be off for some time this sprint. Lets estimate how this affects the delivery this time.”

So the team estimates that since Joe was working on an item in the previous sprint that is directly linked to some items of this sprint, that will have bigger impact on the velocity if we would like to keep the same scope that was planned. Of course, Joe's pair Philip will pick up the work, but it might affect the quality if he is to work alone, and if we pair Philip with, lets say Jane's pair (since her pair is available this sprint for some time) this might slow Philip even further to try to get Jane's pair up to speed. And many other possibilities were discussed, and the scope was readjusted to try to deliver as faster as possible but as close to the highest value stream.

So after some time they estimated that having Joe off for 8 days this sprint and having Jane off for two will trim 5 and 3 (or total of 8) story points of their forecasted velocity for this sprint. We being good project managers, again did our calculations and came up with capacity = avg.velocity - 8 that came up to lets say 60% of the average velocity.

The second approach motivates the team to decide the impact of having people away for some time, bolsters their self management, autonomy and didn't ranked anyone into any category of seniority or general contribution to the team, rather their importance and effect of the following sprint. It may be that next time (if we keep the same vacation duration) Joe might affect 2 points and Jane might affect 5, but it all depends on what the team thinks how and what it will affect.

Also, with the second approach, the team feels more confident to deliver the sprint goal and can better organise towards mitigating the risk. After all, they will be responsible for their forecast, instead of not owing any plan since it was imposed from above (we'll blame the tool).

Buffers as solution…

Is drawing from the backlog Kanban?

Following Scrum, each Product Backlog Item must be "Ready" by the definition of ready agreed by the team to be considered as a candidate for the sprint planning. To get an item to ready it must pass several steps of refinement. Once the team agrees that the item is understood by everyone, is small enough to undertake into one sprint, its testable, negotiable, and so on (see INVEST), the team puts it in the release plan for the following iteration. The team can build up to two or three sprints worth of items as a release plan (and group them according to the highest value order) that can be used as a pool from where they might draw for the next Sprint Planning session.

Lets say that the team completed their Sprint Goal for the current sprint and all of the items in the Sprint Backlog. Now, they can draw from the pool of ready and refined items (or the potential plan for the next sprint) so they can be one step ahead. This will increase their velocity, which is in the spirit of agile, or kaizen. If this improved velocity is stable for at least three sprints, it can be taken as the teams "new" velocity and they can forecast more items in the following itteration.

Why drawing from this "refined backlog" into a sprint is **not** Kanban? In Kanban, we have a ToDo column that we collect new requirements as they come in. We have a work in progress items, limited to some number to follow the lean principles, and once we have an available slot, we pick up things from the ToDo columns. This items here are not refined and are not always ready. So part of the "in progress" action is to get them to a state of ready before we start implementation work on them.

Of course, once the team draws from the "refined backlog" into the "sprint backlog" they can follow the Kanban workflow to move them between states (ex. Ready, In Progress, Code Review, Testing, etc.). But drawing "refined items" from the backlog into the current Sprint, whether they are grouped into a release plan for subsequent sprints or just ordered by highest value, is not Kanban.

Kanban is great for teams that have lots of incoming requests that vary in priority and size. Whereas scrum processes require high control over what is in scope, kanban let's you go with the flow. Kanban is based on a continuous workflow structure that keeps teams nimble and ready to adapt to changing priorities. So if we move the spirit of Kanban outside the constraints of the Sprint, we are now fighting a different battle, that is a beast on its own, SCOPE CREEP.

…and there are also ways to manage that as well.