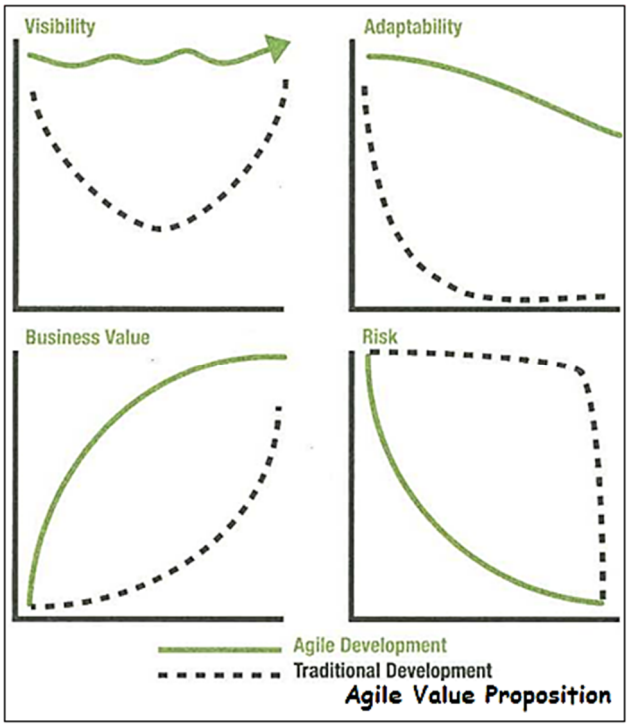
Domain II Value-Driven Delivery

The PMI-ACP Exam consists of 120 questions which can be categorised into seven domains. The second domain: Domain II Value-Driven Delivery is the knowledge about "delivering valuable results by producing high-value increments for review, early and often, based on stakeholder priorities and collecting feedback from the stakeholders on these increments, and using this feedback to prioritize and improve future increments" (source: PMI-ACP Examination Content Outline).
Domain II Value-Driven Delivery accounts for 20% of all questions in the PMI-ACP Exam (i.e. ~24 questions among 120 PMI-ACP Exam questions)



Deliver Value Early (Eat Your Dessert First!):

Deliver the highest-value portions of the project as soon as possible. The reasons are as follows –

1. Firstly, the longer the horizon becomes for risks that can reduce value such as failure, decreased benefits, erosion of opportunities, and so on.
2. Secondly, in this way the team demonstrates an understanding of the stakeholders' needs, shows that they recognize the most important aspects of the project, and proves they can deliver. Tangible results help the team raise the confidence of stakeholders, build rapport with them, and get them onboard early, creating a virtuous circle of support.
3. Thirdly, making decisions that prioritize the value-adding activities and risk reducing efforts for the project, and then executing based on these priorities.

Minimize Waste:

Wasteful activities reduce value. This is why agile methodologies have adopted the lean concept of minimizing waste and other nonvalue-adding activities. 7 wastes are : **Partially done work**, **Extra processes**, **Extra features**, **Task switching**, **Waiting**, **Motion**, **Defects**.

Value-Driven Delivery - 15 tasks grouped within 4 sub-domains:

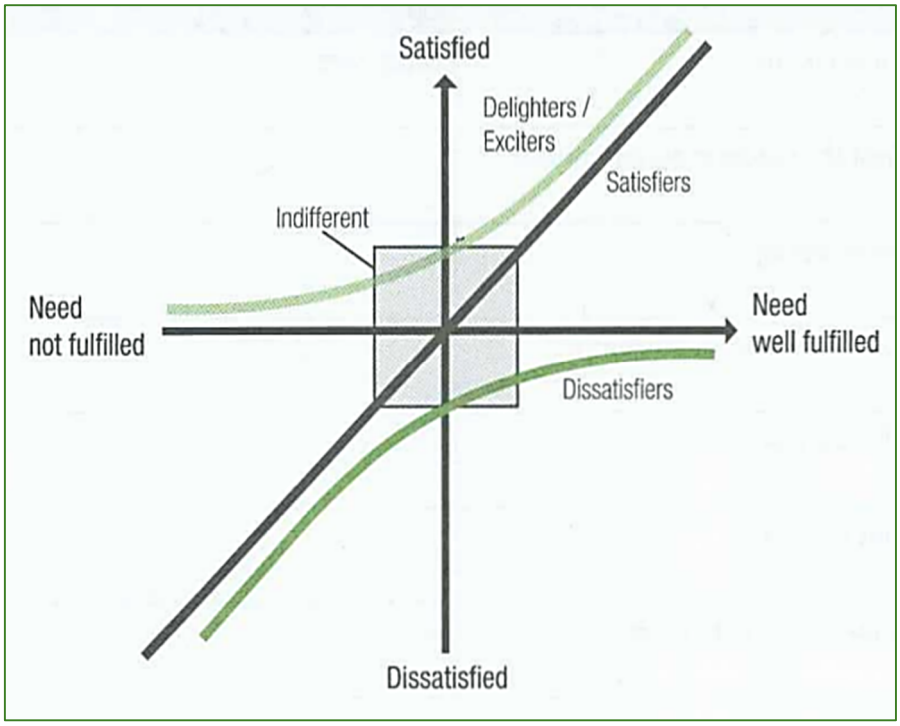
1. **Define Positive Value**
    1. Deliver work **incrementally** to gain competitive advantage and early realization of value.
    2. **Maximize value** delivered to stakeholders while at the same time **minimize non-value-added work**.
    3. Reach consensus on the **acceptance criteria** of the deliverables.
    4. **Refine project processes** based on factors like team experience and organization preferences.
2. **Avoid Potential Downsides (Control Risk)**
    5. Make use of the concept of **Minimally Marketable Features (MMF) / Minimally Viable Product (MVP)** to deliver releasable increments fast.
    6. Solicit **feedback** from stakeholders and review frequently to enhance value.
    7. **Fail fast** by carrying out experiments / spikes early on to reduce risk.
3. **Prioritization**
    8. **Collaborate with stakeholders** to prioritize features in order to realize value early on.
    9. **Review the backlog prioritization** with stakeholder frequently to optimize value delivery.
    10. Identify and prioritize **continuously** the various changing factors affecting the project in order to enhance quality and increase value.
    11. Both value producing and risk reducing work are prioritized in into the backlog in order to **balance value and risks** (non-value).
    12. Non-functional requirements (e.g. security, operations) will need to be considered and prioritized in order to **minimize the likelihood of failure**.

4. **Incremental Development**
    13. **Develop the product incrementally** to reduce risk and deliver value fast.
    14. Inspections, testing and **reviews with stakeholders** are carried out periodically to obtain feedback and make corrections as necessary.
    15. Retrospectives allow overall **improvements** to be made to the project process.

Key Knowledge Factors:

- **Value-driven delivery** is an overarching principle for Agile projects. Projects are carried out to realize values (e.g. economic benefits, competitive advantages, reducing risks, regulatory compliance, etc.)
    - In terms of Agile project management (and the PMI-ACP exam), prioritization is the process where customers **organize / select** product backlog / user stories for implementation based on the perceived **values**
- **Value-based Prioritization** is to organize things so that the most important ones that deliver values are to be dealt with first. **Return on Investment (ROI) / Net Present Value (NPV) / Internal Rate of Return (IRR)** are metrics to assess prioritization based on **monetary values**:
    - ***Return on Investment (ROI) –*** the values a project realized (using present value) compared to the investment; a positive ROI means the project is profitable
    - ***Net Present Value (NPV) –*** the net future cash flow (profit – expenditure) in terms of today's value (adjusted for future inflation, etc.); a positive NPV means the project is profitable
    - ***Internal Rate of Return (IRR) –*** this is somewhat like the interest rate of the investment, the higher the positive IRR, the more profitable the project
- **Customer-valued Prioritization:**
    - Deliver the **highest value** to the customers as early as possible
    - The backlog should be **customer-valued prioritized** while taking into accounts technical feasibilities, risks, dependencies, etc. in order to win customer support
- **Value Prioritization Schemes:**
    - Different agile methodologies use different tools for customer valued prioritization. Scrum has a "**Product Backlog**", FDD has a "**Feature List**", and DSDM has a "**Prioritized Requirements List**" — but the idea is the same.
    - ***Simple schemes –*** rank from high, medium to low (priority 1, 2, 3, ...). It can be problematic since stakeholders have a tendency to designate everything a "Priority 1".
    - ***MoSCoW prioritization scheme –*** Must have, Should have, Could have, would like to have in future (nice to have, but not this time)
    - ***Monopoly money –*** ask customers to give out (fake) money to individual business features in order to compare the relative priority. It can be problematic, if people distributing the money start to question activities, such as documentation, that they perceive as adding little value to the project. The Monopoly money technique is most effective when it's limited to prioritizing business features.
    - ***100-Point method –*** customers are allowed to give, in total 100 points, to various features
    - ***Dot voting / Multi-voting –*** everyone is given a limited number of dots (~20% of the number of all options) to vote on the options. When deciding how many votes to give each person, a good rule of thumb is 20% of the total number of items. So, if there are 40 items to be voted on, we would calculate 40 x 0.2 = 8, and everyone would get 8 votes to distribute.
    - ***Kano analysis –*** plot the features on a graph with axes as Need Fulfilled / Not fulfilled vs Satisfied / Dissatisfied, each feature will then be classified as "**Exciters/Delighter**, **Satisfiers**, **Dissatisfiers**, **Indifferent**". Exciters are of highest values.
        - ✓ ***Delighters/Exciters:*** These features deliver unexpected, novel, or new high-value benefits to the customer. For example, a delighter/exciter feature could be the visual mapping of data that was previously only available in tabular form. Delighters/Exciters yield high levels of customer support.
        - ✓ ***Satisfiers:*** For features that are categorized as satisfiers, the more the better. These features bring value to the customer. For example, a useful reporting function could be a satisfies.

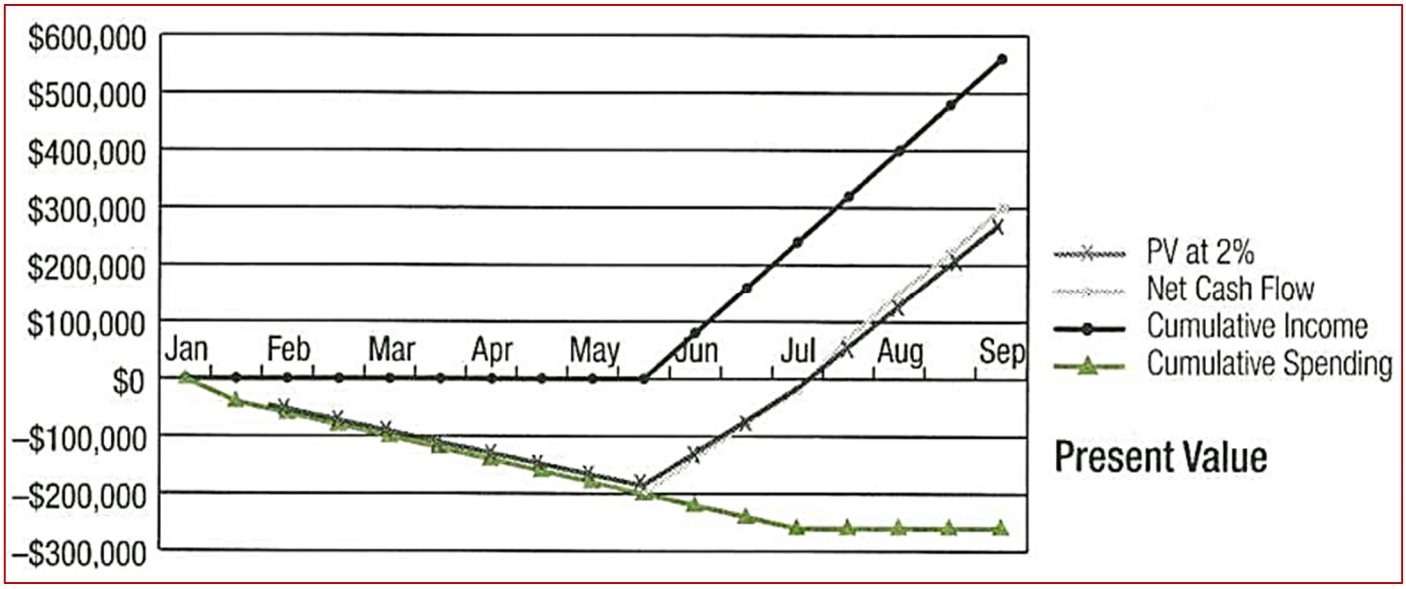
- ✓ ***Dissatisfiers:*** These are things that will cause a user to dislike the product if they are not there but will not necessarily raise satisfaction if they are present. For example, the ability to change a password within the system could be a dissatisfier.
- ✓ ***Indifferent:*** These features have no impact on customers one way or another. Since customers are indifferent to them, we should try to eliminate, minimize, or defer them.



- **Requirements Prioritization Model:**
  - Rate each feature by benefits for having, penalty for not having, cost of producing, risks, etc. and calculate a score using a pre-defined weighted formula
  - **CARVER (Criticality, Accessibility, Return, Vulnerability, Effect, and Recognizability)** relative to the objective and mission of the project
    - ✓ **Criticality –** how important to be done upfront
    - ✓ **Accessibility –** can work on it immediately? or depends on other work / skills?
    - ✓ **Return –**
      - **Return of Investment (ROI) –** The ratio of the benefits received from an investment to the money invested in it, expressed as a percentage.
      - **Net Present Value (NPV) –** The present value of a revenue stream (income minus costs) over a series of time periods
      - **Internal Rate of Return (IRR) –** The discount rate at which the project in flows (revenues) and project outflows (costs) are equal.

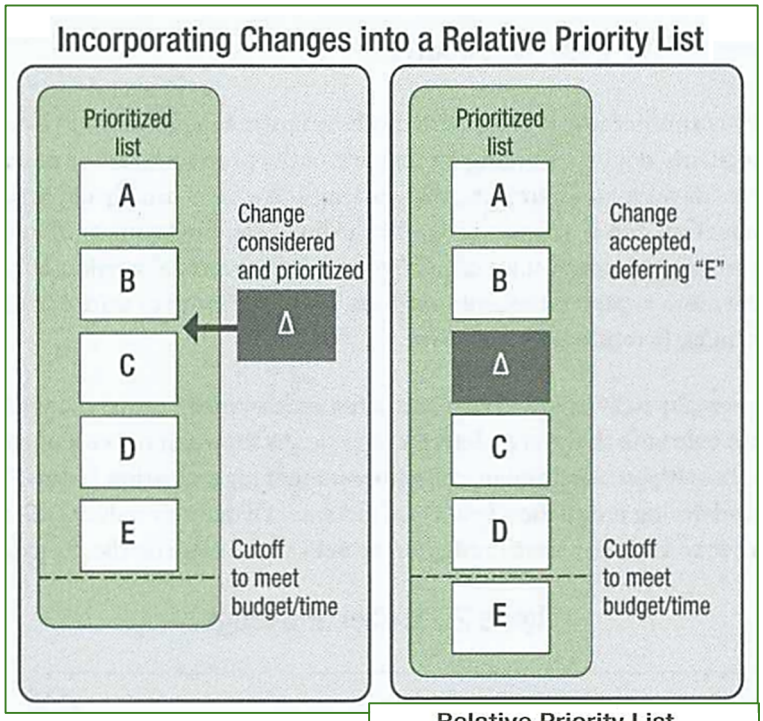
Sample Query:

How to choose between a project that we expect will deliver a 2% ROI in 12 months and one that we expect will deliver a 4% ROI in 36 months? Do inflation and interest over the longer time period cancel out the higher ROI of the second project? To find out, we calculate the NPV of the two projects, and see which one yields more value in today's money.

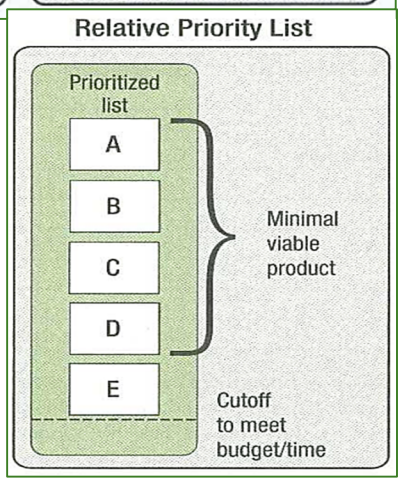


- ✓ **Vulnerability –** how easy to achieve the desired results?
- ✓ **Effect –** what are the effects on the project (help moving towards the goal of the project)?
- ✓ **Recognizability –** have the goals been clearly identified?

- **Relative Prioritization / Ranking:**
  - An ordered list of all user stories / features to be completed with 1 being the highest priority
  - When new features are to be added, it has to be compared, in terms of priority, to all current features
  - The schemes list above can be used to assist the relative prioritization / ranking tasks

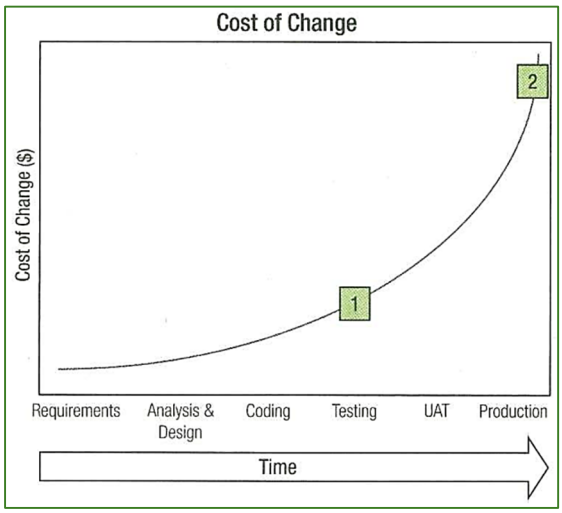


- **Minimally Marketable Features (MMF):**
  - The minimal functionality set (a group of user stories or a package of features) that can deliver values (e.g. useful) to the customers / end-users
  - A distinct and deliverable feature of the system that provides significant values to the customer (can be sold / used immediately)
  - Chosen for implementation after value-based prioritization
  - Can secure Return on Investment instantly



- **Minimal Viable Product (MVP):**
  - The minimal product (with just essential features and no more) that allows can be shipped to early adopters see and learn from the feedback instantly. The concept is somewhat similar to Minimally Marketable Feature (MMF) in which MVP is the first shippable product with the first set of MMF.
- **Compliance:**
  - Conforming to a rule, such as a specification, policy, standard or law (e.g. regulatory compliance)
  - Compliance usually requires documentation, which is somewhat against the principles of agile (working software over documentation)
  - A balance has to be struck (maybe with the help of Agile compliance management systems)
- **Requirements Review:**
  - A group of users and/or recognized experts carrying out reviews on the documented requirements to ensure that:

- ✓ Requirements accurately reflect the needs and priorities of stakeholders
- ✓ Requirements are technical feasible to be fulfilled



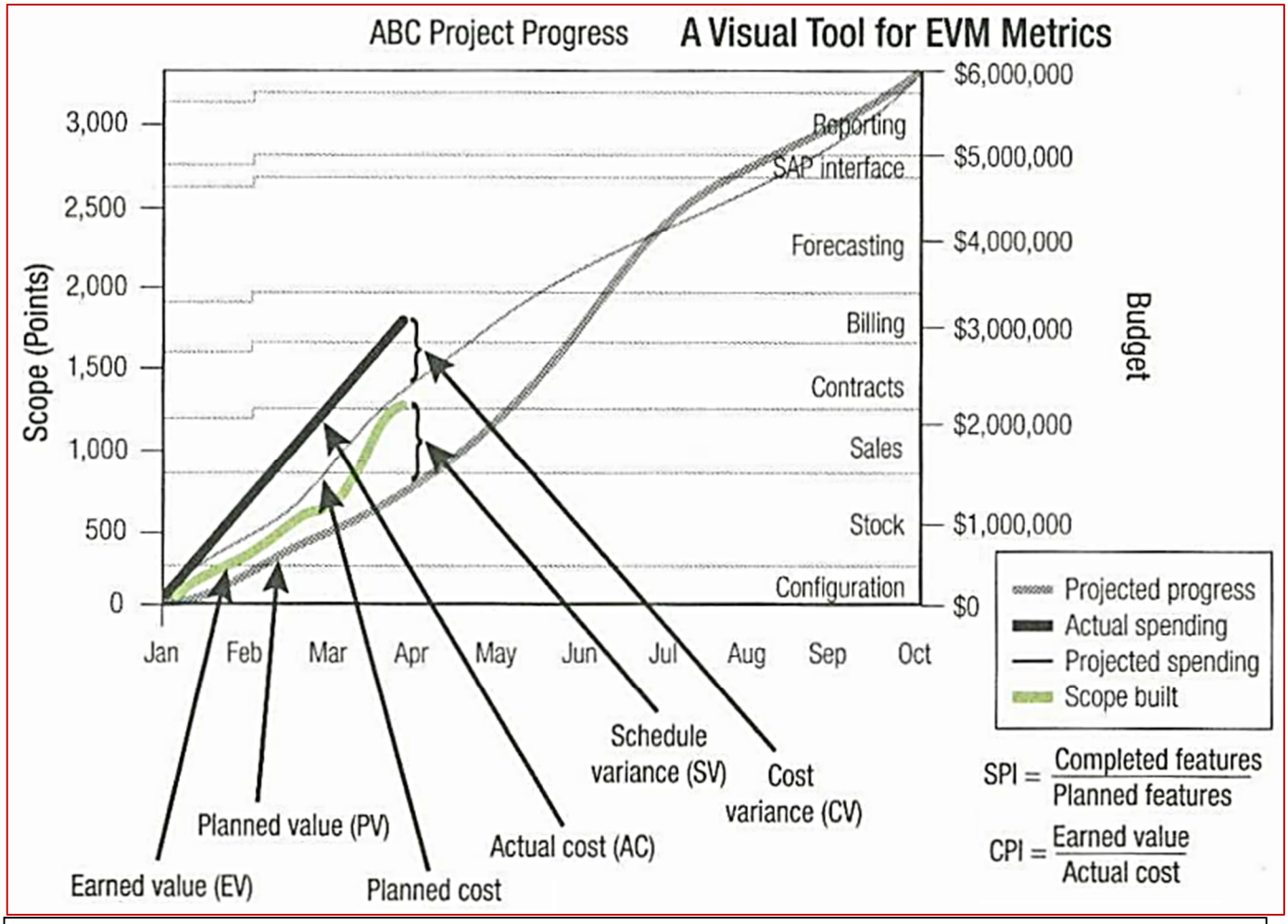
- **Delivering Incrementally:**
  - In the case of software development projects, the working software is usually deployed to a ***test environment for evaluation***, but if it makes sense for the business, the team could deliver functionality directly to ***production in increments***.
  - If we can deliver the "***plain-vanilla***" version of a product or service while working on the more complex elements, we have an opportunity to start realizing the benefits of the product and get an early return on investment.

Automated teller machine (ATM)

MVP	Additional Releases
Dispenses money **Displays balance** **Protects against attack** **Keeps user information secure**	**Accepts cash deposits** **Accepts check deposits** **Remembers user's favorite while withdrawal amounts**

Earned Value Management (EVM) for Agile Projects:

- Earned Value Management is a tool used in traditional project management to measure the progress (in terms of realization of values) of the project
- In Agile, EVM is the measure of **cost performance** of the Agile project though the estimate and actual costs (money spent) can be plotted as a S-curve graph to clearly show whether the project is under- or over-budget, it does not tell whether the progress of the project is ahead of or behind schedule
- Value (story points and money) is calculated at the end of each iteration for work done
  - For the construction of the ***graph for EVM***:
  - The baseline for comparison:
    - ✓ number of planned iterations in a release
    - ✓ planned story points in the release
    - ✓ planned budget for the release
  - Actual measurements:
    - ✓ total story points completed
    - ✓ number of iterations completed
    - ✓ Actual Cost (actual spending) to date
    - ✓ any story points added / removed from the plan (as Agile project requirements are ever-changing)
  - Plotting the chart:
    - ✓ **x-axis:** iterations / date
    - ✓ **y-axis:** i) story points planned; ii) story points completed, iii) planned budget; iv) actual costs
    - ✓ discrepancies between i) and ii) / iii) and iv) reflect the performance of the release
- **Formulas for EVM** (not to be tested on the PMI-ACP exam)
  - Schedule Performance Index (SPI) = Earned Value / Planned Value
  - Cost Performance Index (CPI) = Earned Value / Actual Cost
- EVM does not indicate the quality of the project outcome, i.e. whether the project is successful or not



The above diagram shows how traditional EVM metrics such as **Schedule Performance Index (SPI)** and **Cost Performance Index (CPI)** can be translated into agile terms. For example:

- We planned to complete 30 story points in the last iteration, but we only completed 25 points. To find our SPI, we divide 25 by 30 for an SPI of 0.83. This tells us that we are working at only 83% of the rate planned.
- CPI is the earned value (EV, or value of completed features) to date divided by the actual costs (AC) to date. So, in the above diagram, CPI= $2,200,000 / $2,800,000 = 0.79. This means we are only getting 79 cents on the dollar compared to what we had predicted.

- **Approved Iterations**
  - After the time assigned to an iteration has been used up, the team will hold a review meeting with the stakeholders (mainly management and customers) to demonstrate the working increments from the iteration. If stakeholders approve the product increment against the backlog selected for the iteration, the iteration is said to be approved. Approved iterations may be used in the contract of work as a way to structure the release of partial payment to the contractor.

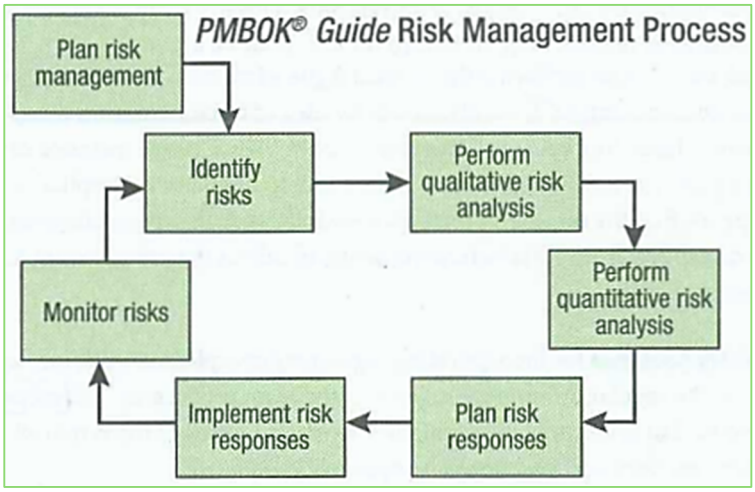
Key Performance Indicators (KPIs):

1. **Rate of progress -** How many features or user stories are getting completed and accepted by the product owner per week or month? So, a simple piece of work might be sized as1 story point, a large chunk of work might be sized as 8 story points, and the project's rate of progress KPI will be expressed in story points per unit of times, such as 20 points per week.
2. **Remaining work -** How much work is left in the backlog? This KPI attempts to quantify the remaining work. We've seen that the backlog items are usually estimated by the development team in story points. And in addition to those estimates, usually some additional effort will be required for unanticipated work, fixes, and ongoing evolution of the product. So, to use an example, let's say that the first half of our project was estimated as 400 points—but once developed, it actually turned out to be 500 points.

3. **Likely completion date -** We look at how much work there is left to do and divide it by our current rate of progress. This is a simple math. I.e. 500 points remaining / 20 points per week = 25 weeks of remaining time, assuming no changes in scope or breaks in the schedule (such as vacations).
4. **Likely costs remaining -** For simple projects, this will be the salary burn rate for the team multiplied by the remaining weeks left. There may also be other fixed costs that we need to consider, such as licenses, equipment, deployment, and training costs, and so on.

Managing Risk:

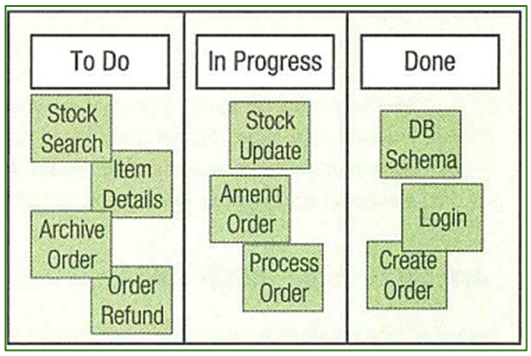
- We can consider, negative project risks (threats) as **anti-value**. Therefore, agile teams not only plan to deliver high-value features early, but they also plan to implement risk avoidance and risk mitigation activities early.



- As for any project, agile teams need to engage the development team, sponsors, customers, and other relevant stakeholders in the process of risk identification. Their ideas, along with the lessons learned from previous projects, risk logs, and industry risk profiles, should all be considered to come up with a complete list of the known and likely risks for the project.
- It's especially important to engage the members of the development team in risk analysis. For one thing, they have unique insights into the risks since they are closer to the technical details. Involving them will also generate increased buy-in for the risk management plan and response actions. If the team is not involved, they have no commitment.
- The primary tools that agile teams use to manage risk are the **risk-adjusted backlog** and **risk burndown charts**.

Agile Low-Tech, High-Touch Tools:

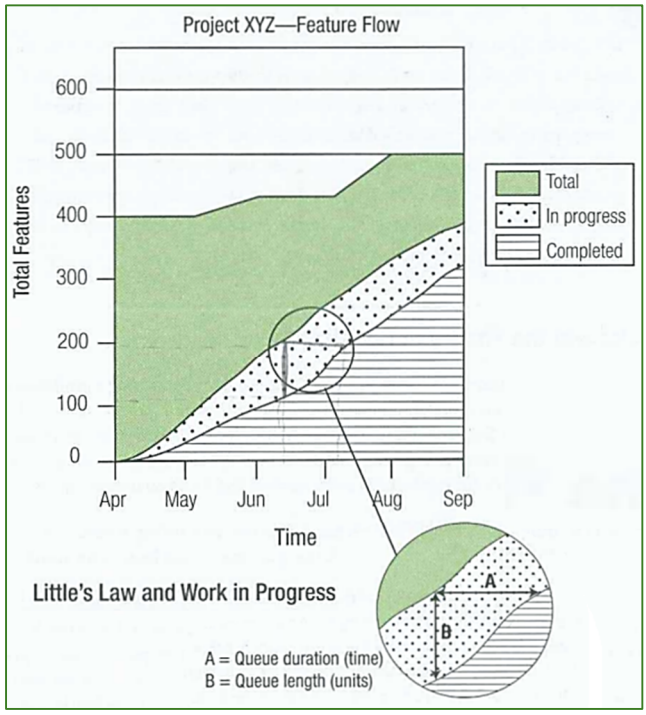
In keeping with the low-tech, high-touch approach, agile teams make widespread use of tangible tools such as task boards, user stories written on 3-by-5-inch index cards, and numbered card decks for planning poker sessions.



Work in Progress (WIP):

Work in Progress (WIP), also sometimes known as "Work in Process" or even "Work in Play", is the term given to work that has been started but has not yet been completed. Excessive levels of WIP are associated with a number of problems, including:

- WIP consumes investment capital and delivers no return on the investment until it is converted into an accepted product. It represents money spent with no return, which is something we want to limit.
- WIP hides bottlenecks in processes that slow over all workflow (or throughput) and masks efficiency issues.
- WIP represents risk in the form of potential rework since there may still be changes to items until those items have been accepted. If there is a large inventory of WIP, there may in turn be a lot of scrap or expensive rework if a change is required.



But the problem is, if we have a bottleneck in processing database requests or designing user interfaces, for example, then tasks may end up sitting for a while, and work accumulates in the system. When this occurs, it is difficult to identify where the bottleneck is, because everyone appears to be busy.

Cumulative Flow Diagrams (CFDs):

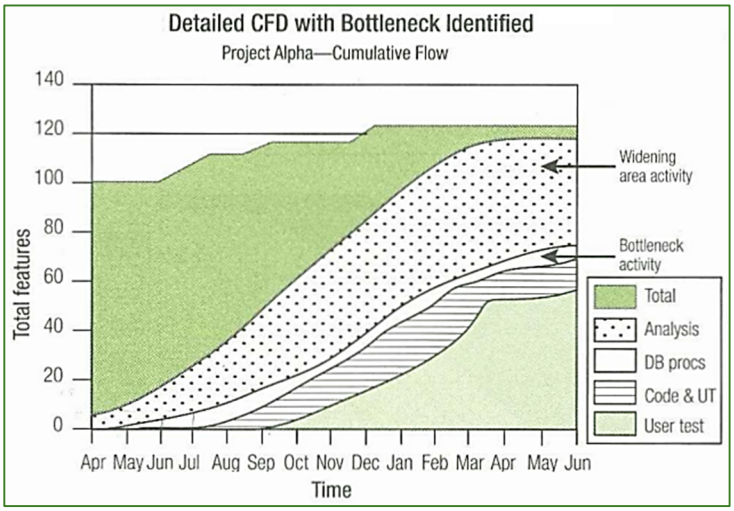
Cumulative flow diagrams (CFDs) are valuable tools for tracking and forecasting the delivery of value.

They can help us gain insight into project issues, cycle times, and likely completion dates.

Basically, CFDs are stacked area graphs that depict the features that are in progress, remaining, and completed over time.

Little's Law is a mathematical formula from queuing theory that can be used to analyse work queues

(i.e., work in progress) on CFDs. This formula proves that the ***duration of a work queue is dependent on its size***, which is why limiting WIP is such a key principle of the Kanban methodology.



Bottlenecks and the Theory of Constraints:

The idea of "**Bottleneck**" is a mathematical concept, a widening area is created when one line is followed by another line of shallower gradient. Since the line gradient indicates the rate of progress for an activity (features over time), a widening area is created above an activity that is progressing at a slower rate. In the figure, we can see that Analysis is a widening area activity, while creating the database stored procedures (DB procs) is a bottleneck activity.

5 Focusing Steps of **Goldratt's Theory of Constraints**, starting with step 2:"

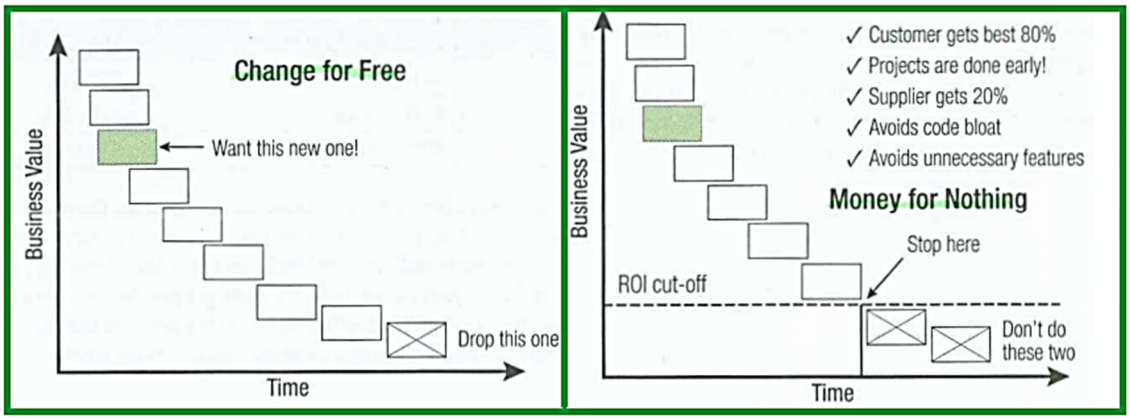
1. Identify the constraint.
2. Exploit the constraint.
3. Subordinate all other processes to exploit the constraint.
4. If after steps 2 and 3 are done, more capacity is needed to meet demand, elevate the constraint.
5. If the constraint has not moved, go back to step 1, but don't let inertia (complacency) become the systems constraint.

Agile Constraints and Contracts:

While agile methods provide great flexibility and allow us to manage changing requirements and priorities, this adaptability and scope flexibility can create problems when defining acceptance criteria for contracts or outsourcing work.

DSDM Contract (Money for Nothing and Change for Free):

Like the "**change for free**" clause, the "**money for nothing**" concept is also only valid if the customer plays their part in the agile project. "**Money for nothing**" allows the customer to terminate the project early when they feel there is no longer sufficient ROI in the backlog to warrant further iterations. The supplier might allow termination of the contract at any time for 20% of the remaining contract value.



Graduated Fixed-Price Contract:

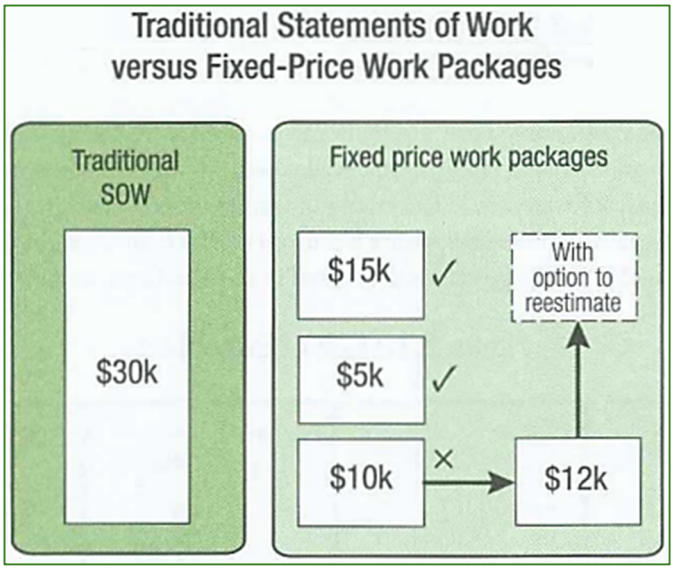
With this kind of contract, both parties share some of the risk and reward associated with schedule variance.

Project Completion	Graduated Rate	Total Fee
Finish early	$110 / hour	$92,000 (Total 836.36 hours)
Finish on time	$100 / hour	$1,00,000 (Total 1,000 hours)
Finish late	$90 / hour	$1,12,000 (Total 1,244 hours)

Fixed-Price Work Packages:

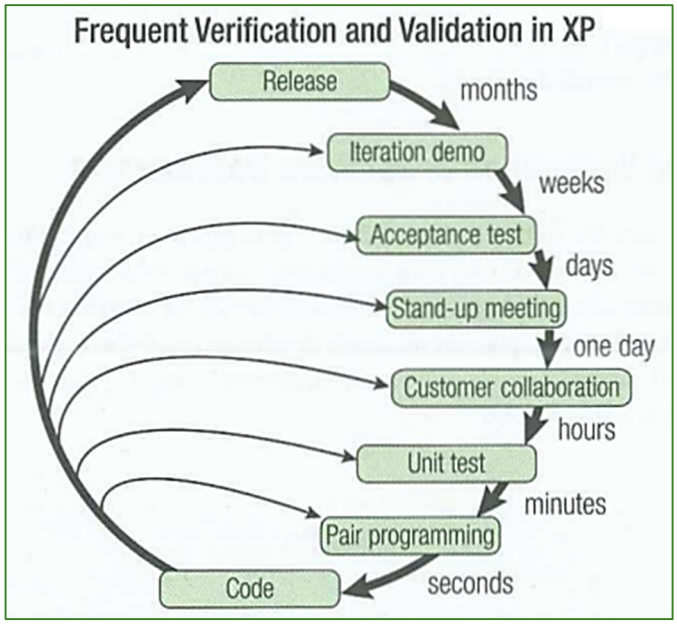
We usually break down **Statements of Work (SOW)** into individual work packages, each with its own fixed price. Then as the work progressed, the supplier was allowed to re-estimate the remaining work packages in the statements of work based on new information and risks.

Using **Fixed-Price Work** packages allows the customer to reprioritize the remaining work based on evolving costs. It also gives the supplier the ability to update their costs as new details emerge, removing the need for the supplier to build excess contingency funds into the project cost.



Verifying and Validating Value:

What one person describes is often very different from how the listener interprets it. This semantic gap is called the "***Gulf of Evaluation***".



Frequent Verification and Validation:

Like the old saying, "***A stitch in time saves nine***", agile uses regular testing, checkpoints, and reviews to address problems before they get bigger.

Exploratory and Usability Testing:

Exploratory testing differs from scripted testing that attempts to exercise all the functional components of a system; instead, it relies on the tester's autonomy, skill, and creativity in trying to discover issues and unexpected behaviour.

Usability testing attempts to answer the question, "How will an end user respond to the system under realistic conditions?" The goal of this kind of testing is to diagnose how easy it is to use the system, and help uncover where there are problems that might need redesign or changes.

Continuous Integration:

Source code control system: This is the software that performs version control on all the files that represent the product being developed.

Build tools: The source code needs to be compiled before tests can be run. Most integrated development environments (IDEs) serve as a build tool to compile the code.

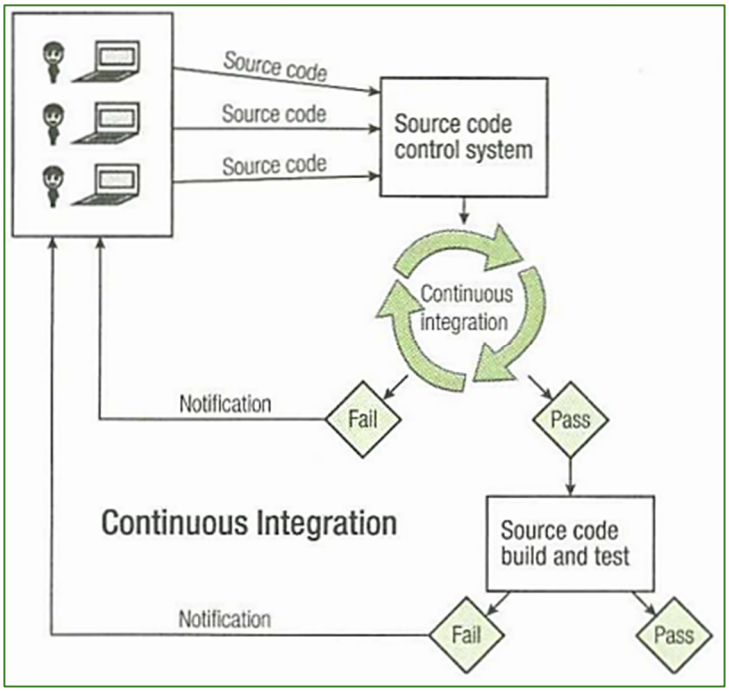
Test tools: As part of the build process, unit tests are run to ensure that the basic functionality operates As planned. Unit test tools execute the small, atomic tests that are written in these tools to check the code for unanticipated changes in behaviour.

Scheduler or trigger: Builds might be launched on a regular schedule (such as every hour) or every time the system detects a change to the source code.

Notifications: If a build fails, the team needs to be notified so they can correct the build as soon as possible. These notifications may be sent via e-mail or instant messaging.

Pros and Cons of Continuous Integration:

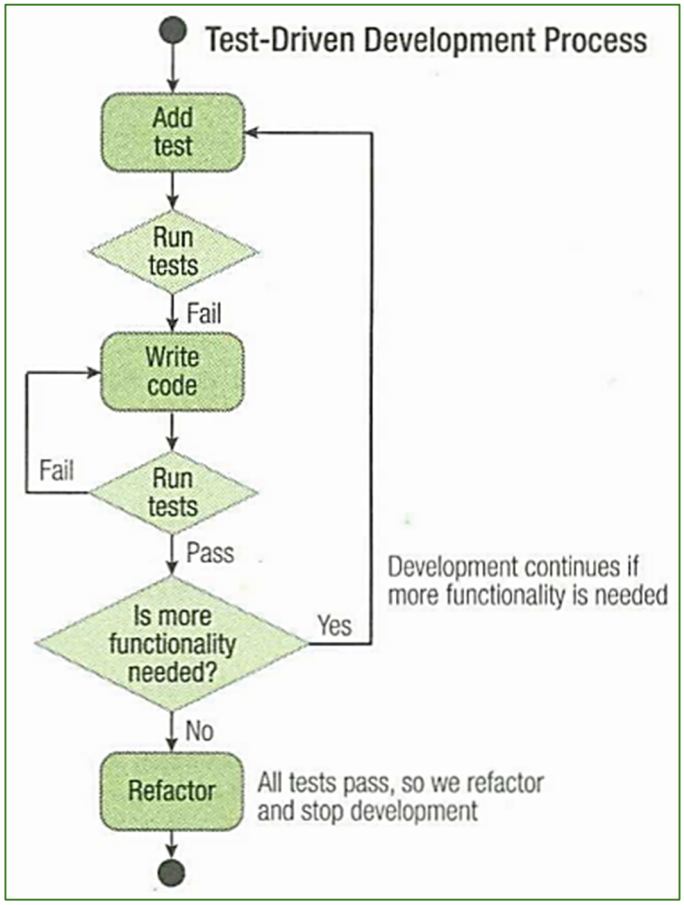
Continuous integration is important for agile projects because it provides the following benefits:



- The team receives an **early warning** of broken, conflicting, or incompatible code.
- Integration problems are **fixed as they occur**, rather than as the release date approaches. This moves any related changes **down the cost of change** curve and avoids last-minute work before releases.
- The team receives **immediate feedback** on the system-wide impacts of the code they are writing.
- This practice ensures **frequent unit testing** of the code, alerting the team to issues sooner rather than later.
- If a problem is found, the code can be **reverted back to the last known bug-free state** for testing, demo, or release purposes.

The disadvantages or costs of using continuous integration are:

- The setup time required to establish a build server machine and configure the continuous integration software; this is typically done as an **iteration 0 activity** (before the development work begins on the project)
- The **cost of procuring a machine** to act as the build server, since this is usually a dedicated machine
- The **time required** to build a suite of **automated**, comprehensive tests that run whenever code is checked in



Test-Driven Development (TDD):

The philosophy behind TDD is that tests should be written before the code is written. In other words, developers should first think about how the functionality should be tested and then write tests in a unit testing language (such as NUnit or JUnit) before they actually begin developing the code. Initially the tests will fail since developers have not yet written the code to deliver the required functionality.

Red, Green, Refactor / Red, Green, Clean:

The process of writing a test that initially fails, adding code until the test passes, and then refactoring the code is known as "**Red, Green, Refactor**" or sometimes "**Red, Green, Clean**".

Acceptance Test-Driven Development (ATDD):

In ATDD there are 4 phases –

- **Discuss the requirements:** During the planning meeting, we ask the product owner or customer questions that are designed to gather acceptance criteria.
- **Distill tests in a framework-friendly format:** In this next phase, we get the tests ready to be entered into our acceptance test tool. This usually involves structuring the tests in a table format.
- **Develop the code and hookup the tests:** During development, the tests are hooked up to the code and the acceptance tests are run. Initially the tests fail because they cannot find the necessary code. Once the code is written and the tests are hooked to it, the tests validate the code. They may again fail, so the process of writing code and testing continues until the code passes the tests.
- **Demo:** The team does exploratory testing using the automated acceptance testing scripts, and demos the software.