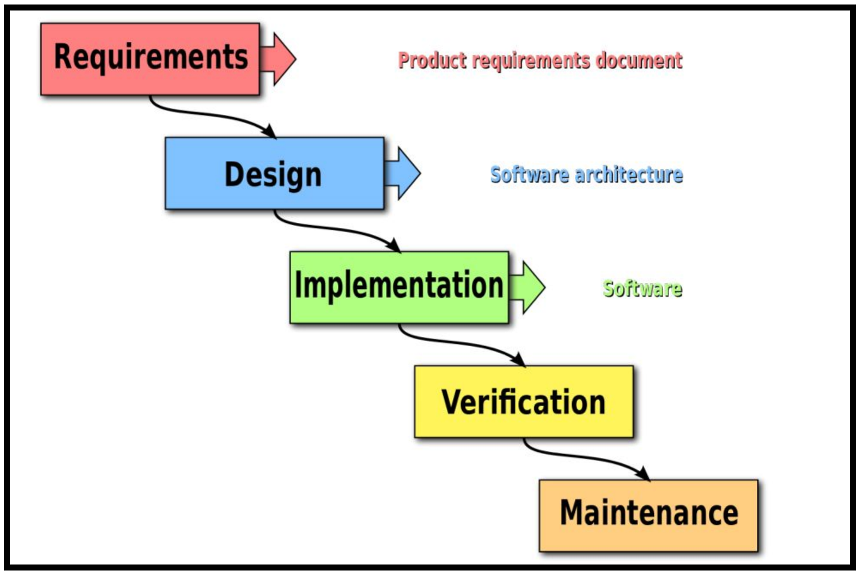
Software Development Life Cycle (SDLC):

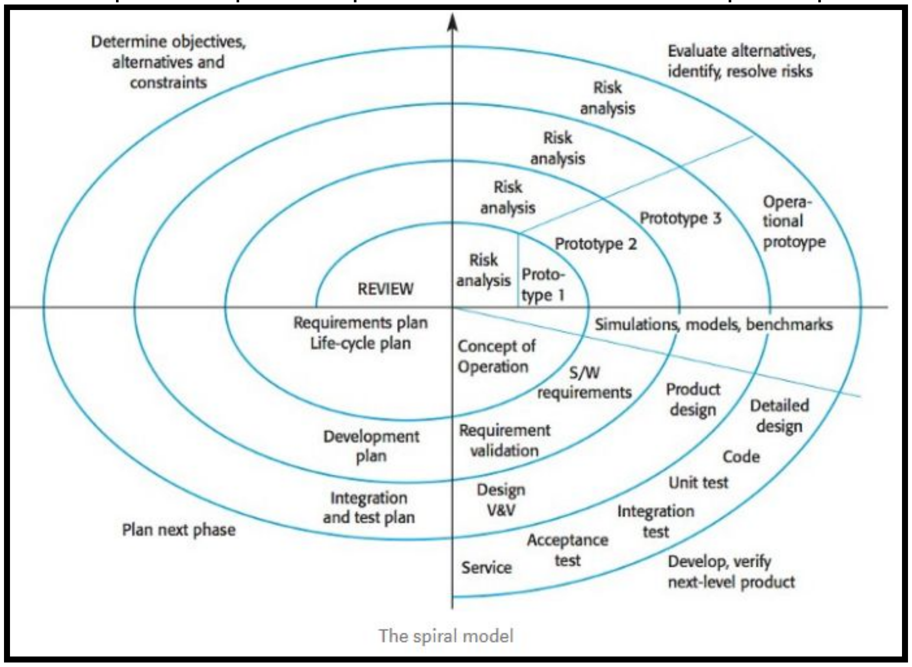
- **Software Process:**
- It is:
    - A structured set of activities, used by a team to develop software systems.
    - The standards, practices, and conventions of a team.
    - A description of how a team performs its work.
- Some synonyms of software process are:
    - Software Development Process
    - Software Development Methodology
    - Software Development Life Cycle
- In a nutshell, it is a structured description of how a software development team goes through.
  I.e.
  Deciding what to build.
  Building it.
  Deciding if they like what they built.
- A **software process** is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch, or, modifying an existing system.
- Over time, people develop new software process models.
- A **software process model** represents the order in which the activities of software development will be undertaken. It describes the sequence in which the phases of the software lifecycle will be performed.
  A model is a template/description of a process.
  Different processes can implement the same model.
  Think of “Process vs. Model” like “Class vs. Interface”
- Examples of software process models are:
    - Waterfall Model
    - Prototyping Model
    - Spiral Model
- **Software Process Model vs. Software Process:**
- Software process is a coherent set of activities for specifying, designing, implementing and testing software systems.
- A software process model is an abstract representation of a process that presents a description of a process from some particular perspective.
- There are many different software processes but all involve:
    - **Specification:** Defining what the system should do.
    - **Design and Implementation:** Defining the organization of the system and implementing the system.
    - **Validation:** Checking that it does what the customer wants.
    - **Evolution:** Changing the system in response to changing customer needs.
- **Waterfall Method:**
- The **waterfall model** is a sequential (non-iterative) design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of:
    1. Requirements analysis
    2. Design
    3. Implementation
    4. Testing (verification)
    5. Maintenance

- I.e.



- The result of each phase is one or more documents that should be approved and the next phase shouldn't be started until the previous phase has completely been finished. I.e. Each phase is carried out completely, for all requirements, before proceeding to the next. Furthermore, this process is strictly sequential. No backing up or repeating phases.
- The waterfall model should only be applied when requirements are well understood and unlikely to change radically during development as this model has a relatively rigid structure which makes it relatively hard to accommodate change when the process is underway.
- Pros:
    - Time spent early in the software production cycle can reduce costs at later stages.
    - Suitable for highly structured organizations.
    - It places emphasis on the documentation, contributing to corporate memory.
    - It provides a structured approach; the model itself progresses linearly through discrete, easily understandable and explainable phases and thus is easy to understand.
    - It also provides easily identifiable milestones in the development process.
    - It is well suited to projects where requirements and scope are fixed, the product itself is firm and stable, and the technology is clearly understood.
    - Simple, easy to understand and follow.
    - Highly structured, therefore good for beginners.
    - After specification is complete, low customer involvement is required.
- Cons:
    - Inflexible and can't adapt to changes in requirements.
    - Only good for projects where the requirements are already laid out and won't change too much.
- **Spiral Model:**
- The spiral model is a risk-driven software development process model which was introduced for dealing with the shortcomings in the traditional waterfall model. I.e. The spiral model is a risk-driven model where the process is represented as a spiral rather than a sequence of activities. Furthermore, it was designed to include the best features from the waterfall and prototyping models, and introduces a new component: risk-assessment.

- Each loop of the spiral is a phase of the software development process.



The spiral model

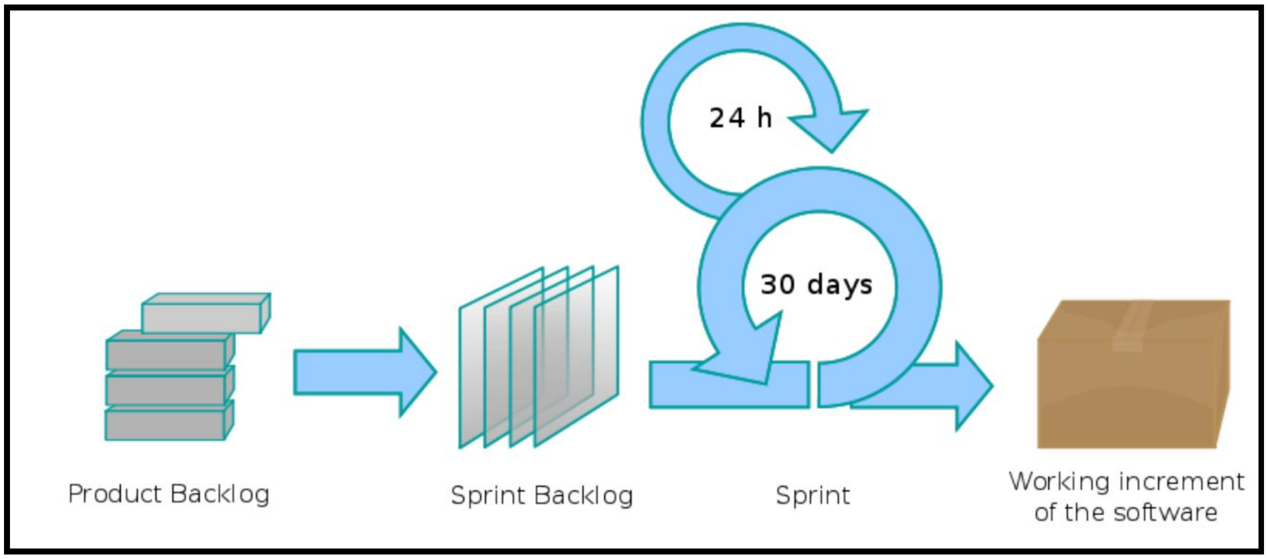
- Each loop in the spiral is split into four sectors:
    1. **Objective setting**: The objectives and risks for that phase of the project are defined.
    2. **Risk assessment and reduction:** For each of the identified project risks, a detailed analysis is conducted, and steps are taken to reduce the risk. For example, if there's a risk that the requirements are inappropriate, a prototype may be developed.
    3. **Development and validation:** After risk evaluation, a process model for the system is chosen. So if the risk is expected in the user interface then we must prototype the user interface. If the risk is in the development process itself then use the waterfall model.
    4. **Planning:** The project is reviewed and a decision is made whether to continue with a further loop or not.
- Pros:
    - Manages uncertainty inherent in exploratory projects.
- Cons:
    - Difficult to establish stable documents; things keep getting modified during each iteration.
- The spiral model has been very influential in helping people think about iteration in software processes and introducing the risk-driven approach to development. In practice, however, the model is rarely used.
- **Prototype Model:**
- A prototype is a version of a system or part of the system that is developed quickly to check the customer's requirements or feasibility of some design decisions.
- A prototype is useful when a customer or developer is not sure of the requirements, or of algorithms, efficiency, business rules, response time, etc.
- In prototyping, the client is involved throughout the development process, which increases the likelihood of client acceptance of the final implementation.
- In prototyping, the project is done in cycles.

- In each cycle, the sequence of phases is:
    1. Gather basic requirements.
    2. Implement prototype (e.g. UI with all the functionality mocked/faked).
    3. Collect feedback from users.
    4. Adjust
- Pros:
    - At the end of each cycle, the team can assess their work, adjust to new requirements, and improve its work.
    - More interaction with users helps us ensure that we are building the right product.
- Cons:
    - Building a prototype is not the same as building a product. This model ignores a big chunk of the system that is not trivial to implement.
    - You do not always have patient/forgiving users who will be willing to try your prototypes.
- **Alternative Methodologies:**
- As companies began to realize that the waterfall method was failing them (large projects were failing completely or going way over budget) alternatives were sought.
- A gradual trend was in methods that used an incremental development of product using iterations (almost like smaller waterfalls).
- Iterations could be only a few weeks, but still included the full cycle of analysis, design, coding, etc.
- These various lightweight development methods were later referred to as **agile methodologies**.
- As our models evolve, they encourage software development teams to:
    - Be more flexible and adaptive to changing requirements.
    - Collect feedback from users more frequently.
    - Release code more frequently.
- And then came the term Agile.
- Agile is neither a process nor a model but is a term that describes a process, model, or a team. Essentially, it means “Flexible and adaptive process/team, suitable for projects with constantly changing requirements".
- **Agile Manifesto:**
- We value:
    - **Individuals and interactions** over **processes and tools**.
    - **Working software** over **comprehensive documentation**.
    - **Customer collaboration** over **contract negotiation**.
    - **Responding to change** over **following a plan**.
- There is value to the items on the right, but the left is valued more.
- **Agile:**
- Agility is flexibility. It is a state of dynamic, adapted to the specific circumstances.
- Agile refers to a number of different frameworks that share these values.
  I.e. Agile is an umbrella term for a set of methods and practices based on the values and principles expressed in the Agile Manifesto that is a way of thinking that enables teams and businesses to innovate, quickly respond to changing demand, while mitigating risk.
- Examples of agile frameworks are:
    - Test Driven Development (TDD)
    - Extreme Programming (XP)
    - Scrum
    - Lean Software

- **Test Driven Development (TDD):**
- A concept that started in the late 90s.
- Used (to some extent) by many Agile teams.
- The idea is to write the tests, before you write the code.
- The tests are the requirements that drive the development.
- A software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and make code simple and bug-free.
- TDD ensures that your system actually meets requirements defined for it. It helps to build your confidence about your system.
- In TDD more focus is on production code that verifies whether testing will work properly. In traditional testing, more focus is on test case design. Whether the test will show the proper/improper execution of the application in order to fulfill requirements.
- In TDD, you achieve 100% coverage test. Every single line of code is tested, unlike traditional testing.
- Traditionally, TDD means:
    - Write a failing test.
    - Write the (least amount of) code to pass the test.
    - Repeat.
    - Every now and then refactor/cleanup code.
- However, in practice, each team decides when and where it makes sense for tests to drive development.
- Most (good) teams borrow some of the concepts of TDD, such as the fact that tests are used as specification/documentation and the fact that we should automate tests in fragile/crucial areas of your system.
- TDD makes sense for agile teams:
    - Agile is about “moving fast".
    - If your code is easy to test, it is usually also easy to build/deploy automatically.
    - Being able to automate your tasks helps your team move fast.
- Software development teams can adopt TDD with different types of testings such as:
    - **Unit Test**
    - **Integration:** Test that all the different units in the system play nicely together.
    - **Acceptance:** Specify customer’s requirement.
    - **Regression:** Verify we didn’t break anything that was working before. Automated unit tests can be used as regression tests.
- Advantages of TDD:
    - Early bug notification:
        - Using TDD, over time, a suite of automated tests is built up that you and any other developer can rerun at will.
    - Better designed, cleaner and more extensible code:
        - TDD helps developers understand how the code will be used and how it interacts with other modules.
        - It results in better design decisions and more maintainable code.
        - TDD allows writing smaller code having single responsibility rather than monolithic procedures with multiple responsibilities. This makes the code simpler to understand.
        - TDD also forces you to write only production code to pass tests based on user requirements.

    - Confidence to refactor:
        - If you refactor code, the code might break. By having a set of automated tests, you can fix those bugs before release.
        - Using TDD should result in faster, more extensible code with fewer bugs that can be updated with minimal risks.
    - Good for teamwork:
        - In the absence of any team member, other team members can easily pick up and work on the code. It also aids knowledge sharing, thereby making the team more effective overall.
    - Good for developers:
        - Though developers have to spend more time writing TDD test cases, it takes a lot less time for debugging and developing new features. You will write cleaner, less complicated code.
- **Extreme Programming (XP):**
- XP is a model that was getting a lot of hype in the late 90s.
- XP is an Agile model, consisting of many rules/practices, one of which is TDD.
- XP is a very detailed model, but in practice, most teams adopt a subset of its rules.
- Some highlights of XP:
    - Iterative incremental model.
    - Better teamwork.
    - Customer's decisions drive the project.
    - Dev team works directly with a domain expert.
    - Accept changing requirements, even near the deadline.
    - Focus on delivering working software instead of documentation.
- A key assumption of XP is that the cost of changing a program can be held mostly constant over time. This can be achieved with:
    - Emphasis on continuous feedback from the customer
    - Short iterations
    - Design and redesign
    - Coding and testing frequently
    - Eliminating defects early, thus reducing costs
    - Keeping the customer involved throughout the development
    - Delivering working product to the customer
- Extreme Programming involves:
    - Writing unit tests before programming and keeping all of the tests running at all times. The unit tests are automated and eliminate defects early, thus reducing the costs.
    - Starting with a simple design just enough to code the features at hand and redesigning when required.
    - Programming in pairs, called **pair programming**, with two programmers at one screen, taking turns to use the keyboard. While one of them is at the keyboard, the other constantly reviews and provides inputs.
    - Integrating and testing the whole system several times a day.
    - Putting a minimal working system into the production quickly and upgrading it whenever required.
    - Keeping the customer involved all the time and obtaining constant feedback.

- **Scrum:**



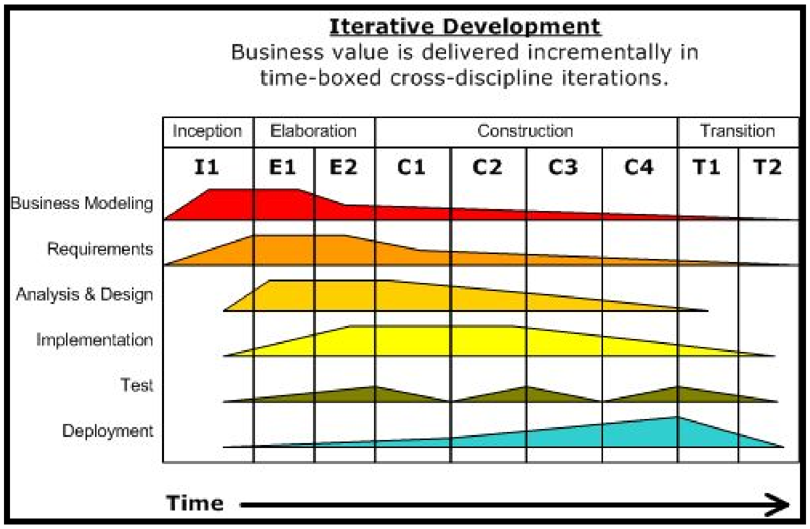
- **Scrum** is a flexible, holistic product development strategy where a development team works as a unit to reach a common goal. Scrum is mainly about the management of software development projects.
- **Sprint:** The actual time period when the scrum team works together to finish an increment. Two weeks is a pretty typical length for a sprint, though some teams find a week to be easier to scope or a month to be easier to deliver a valuable increment. During this period, the scope can be re-negotiated between the product owner and the development team if necessary. This forms the crux of the empirical nature of scrum. Key features of sprints:
    - It is a basic unit of development in a scrum.
    - It is of fixed length, typically from one week to a month.
    - Each sprint begins with a **planning meeting** to determine the tasks for the sprint and estimates are made.
    - During each sprint a potentially deliverable product is produced.
    - Features are pulled from a **product backlog**, a prioritized set of high level work requirements.
- **Sprint planning:** The work to be performed during the current sprint is planned during this meeting by the entire development team. This meeting is led by the **scrum master** and is where the team decides on the sprint goal. Specific **user stories** are then added to the sprint from the product backlog. These stories always align with the goal and are also agreed upon by the scrum team to be feasible to implement during the sprint. At the end of the planning meeting, every scrum member needs to be clear on what can be delivered in the sprint and how the increment can be delivered.
- **User Story:** An informal, general explanation of a software feature written from the perspective of the end user or customer. The purpose of a user story is to articulate how a piece of work will deliver a particular value back to the customer. User stories are a few sentences in simple language that outline the desired outcome. They don't go into detail. Requirements are added later, once agreed upon by the team.
- **Product Backlog:** The master list of work that needs to get done maintained by the product owner or product manager. This is a dynamic list of features, requirements, enhancements, and fixes that acts as the input for the sprint backlog. It is, essentially, the team’s “To Do” list. The product backlog is constantly revisited, re-prioritized and

maintained by the Product Owner because, as we learn more or as the market changes, items may no longer be relevant or problems may get solved in other ways.

- **Sprint Backlog:** The list of items, user stories, or bug fixes, selected by the development team for implementation in the current sprint cycle. Before each sprint, in the sprint planning meeting the team chooses which items it will work on for the sprint from the product backlog. A sprint backlog may be flexible and can evolve during a sprint.
- **Increment/Sprint Goal:** The usable end-product from a sprint.
- **Daily Scrum/Daily Standup:** A short meeting that happens at the same place and time each day. At each meeting, the team reviews work that was completed the previous day and plans what work will be done in the next 24 hours. This is the time for team members to speak up about any problems that might prevent project completion. Some features of the daily scrum:
    - No more than 15 minutes.
    - Meetings must start on-time, and happen at the same location.
    - Each member answers the following:
        - What have you done since yesterday?
        - What are you planning on doing today?
        - Are there any impediments or stumbling blocks?
    - A scrum master will handle resolving any impediments outside of this meeting
- **Scrum Master:** The person on the team who is responsible for managing the process, and only the process. They are not involved in the decision-making, but act as a lodestar to guide the team through the scrum process with their experience and expertise. The scrum master is the team role responsible for ensuring the team follows the processes and practices that the team agreed they would use.
- In traditional agile development software is brought to release level every few months. **Releases**, which are sets of sprints, are used to produce shippable versions of software products.
- A key feature of scrum is that during a project a customer may change their minds about what they want/need. We need to accept that the problem cannot be fully understood or defined and instead allow teams to deliver quickly and respond to changes in a timely manner.
- **Lean Software Development:**
- Lean software development is a concept that emphasizes optimizing efficiency and minimizing waste in the development. It is an agile framework based on optimizing development time and resources, eliminating waste, and ultimately delivering only what the product needs.
- Lean software development is an agile framework sharing the following values:
    - Eliminate waste
    - Amplify learning
    - Decide as late as possible
    - Deliver as fast as possible
    - Empower the team
    - Build flexibility in
    - See the whole
- **Agile Unified Process (AUP):**
- AUP is an iterative-incremental process consisting of seven sub-processes or workflows:
    1. **Model:** Understand the business, the problem domain, and try to identify a viable solution.
    2. **Implementation:** Transform your model(s) into executable code and perform unit testing.

3. **Test:** Perform an objective evaluation to ensure quality.
4. **Deployment:** Plan and execute the delivery of the system.
5. **Configuration Management:** Manage access to your project artifacts.
6. **Project Management:** Direct the activities that take place on the project.
7. **Environment:** Support the rest of the effort by ensuring that the proper resources are available for the team as needed.

- The AUP workflows go through four phases:
    1. **Inception:** The goal is to identify the initial scope of the project, a potential architecture for your system, and to obtain initial project funding and stakeholder acceptance.
    2. **Elaboration:** The goal is to prove the architecture of the system.
    3. **Construction:** The goal is to build working software on a regular, incremental basis which meets the highest-priority needs of your project stakeholders.
    4. **Transition:** The goal is to validate and deploy your system into your production environment.



- **Relationship with Customers:**
- Customer Specific:
    - One customer with a specific problem.
    - The customer may be another company with a contractual agreement or a division within the same company.
- Market-based:
    - Selling the product to a general market.
    - In some cases, the product must generate customers.
    - The marketing team may act as a substitute customer.
- Community-based:
    - Intended as a general benefit to some community.
    - Examples include open-source tools and tools for scientific research.
- Hybrid:
    - A mix of the above.

Project Planning:

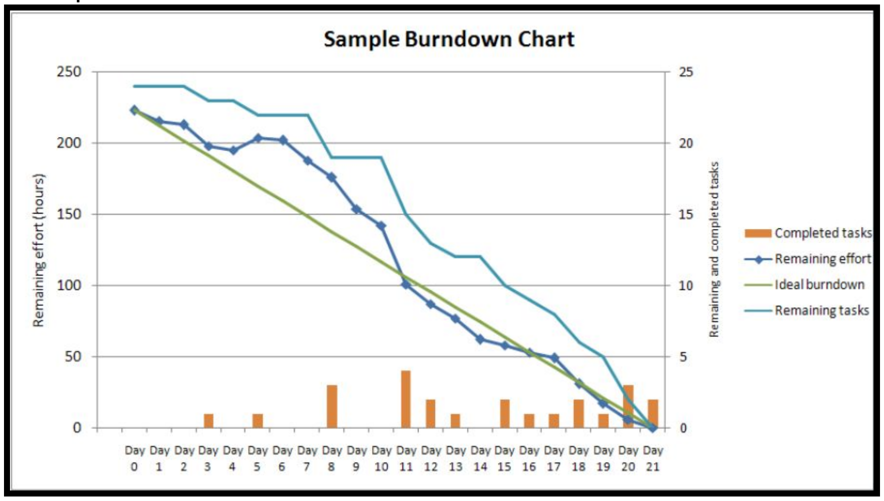
- **Parts of Project Planning:**
- Given:
    - A list of customer requirements.
    - Examples include a set of use cases or a set of change requests.
- Estimate:
    - How long each one will take to implement (cost).
    - How important each one is (value).
- Plan:
    - Which requests should be included in the next release.
- Complication:
    - Customers care about other stuff, such as security and quality, too.
- **Key Principles of Management:**
- A manager can control 4 things:
    1. Resources - Can get more money, personnel, etc
    2. Time - Can vary the schedule and delay milestones
    3. Product - Can vary the amount of functionality
    4. Risk - Can decide which risks are acceptable
- Approach:
    - Understand the goals and objectives.
    - Understand the constraints.
    - Plan to meet the objectives within the constraints.
    - Monitor and adjust the plan.
    - Preserve a calm, productive, positive work environment.
- **Note:** You cannot control what you cannot measure.
- **Strategies:**
- Fixed Product:
    1. Identify the customer requirements.
    2. Estimate the size of software needed to meet them.
    3. Calculate the time required to build the software.
    4. Get the customer to agree to the cost and schedule.
- Fixed Schedule:
    1. Fix a date for the next release.
    2. Obtain a prioritized list of requirements.
    3. Estimate the effort for each requirement.
    4. Select requirements from the list until there's no more left or you don't think you can work on more tasks.
- Fixed Cost:
    1. Agree with the customer on how much they wish to spend.
    2. Obtain a prioritized list of requirements.
    3. Estimate the cost for each requirement.
    4. Select requirements from the list until the "cost" is used up.
- **Estimating Effort - Constructive Cost Model (COCOMO):**
- Predicts the cost of a project from a measure of size (lines of code).
- The basic model is: $E = aL^b$
  E = effort
  a & b = project specific folders
  L = lines of code
- Modelling process:
    1. Establish the type of project (organic, semidetached, embedded, etc).
       This gives sets of values for a and b.

2. Identify the component modules and estimate L for each module.
3. Adjust L according to how much code is reused.
4. Compute E using the formula above.
5. Adjust E according to the difficulty of the project.
6. Compute time using $T = cE^d$, where c and d are provided for different project types, like a and b.

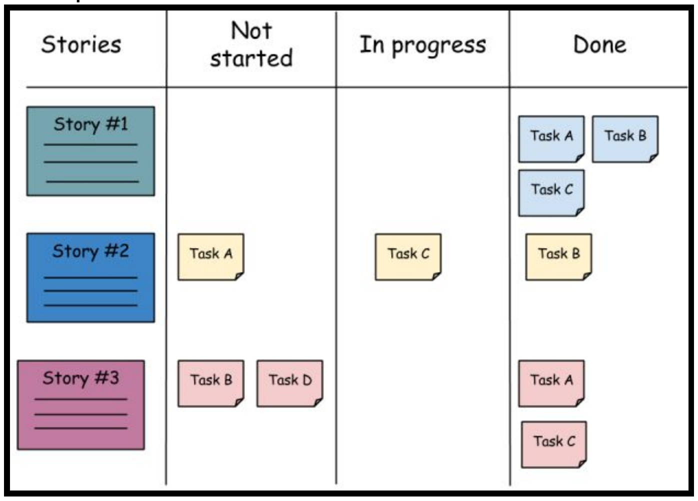
- **Estimating Size - Function Points:**
- Used to calculate the size of software from a statement of the problem.
- Tries to address the variability in lines of code estimates used in models such as COCOMO.
- Basic model is: $FP = a_1I + a_2O + a_3E + a_4L + a_5F$
- Each a_i is a weighting factor for their respective metric.
- I is the number of user inputs (data entry).
- O is the number of user outputs (reports, screens, error messages).
- E is the number of user queries.
- L is the number of files.
- F is the number of external interfaces.
- **Three-Point Estimating:**
- W = worst possible case
- M = Most likely case
- B = Best possible case
- $E = \sum_i \frac{Wi + 4Mi + Bi}{6}$
- **Story Points:**
- We need a common way to compare story sizes.
- It can be hard to find common ground between a programming story and a database management story.
- **Story points** are a relative measure of a feature's size or complexity.
  They are not durations nor a commitment to when a story will be completed.
  Different teams have different velocities, so they may complete stories at different rates depending on experience.
- A good tool to do the estimation is **planning poker**. It is a series similar to the Fibonacci Series that can be a useful range for story points. Here, each number is almost the sum of the two preceding numbers: 0, 1, 2, 3, 5, 8, 13, 20, 40, 100.
- 0-points estimates are used for trivial tasks that require little effort, though too many zero-pointers can add up.
- Only use numbers within the set and avoid averages. We avoid averages because if a user story turns out to be harder than expected, then the people who picked a higher number will say "I told you" to the people who picked a lower number. The average does not convince other people.
- What happens is this:
  1. A feature is mentioned.
  2. Each person in the team takes a number from the set, but doesn't show/tell anyone else yet.
     They choose the number based on how difficult they think implementing the feature will be.
     A feature with point 0 means that it requires very little effort.
  3. After 3 seconds, everyone shows their number.
  4. If everyone or most people have the same number, it's good.

5. If everyone or most people have different numbers, then each person has to defend why they picked their number.
   Once the discussion has been carried out, there is a second round of voting.
6. The process repeats until everyone agrees.
   If people consistently do not agree with one another, then the user story is not a good user story. This is because one of the features of a user story is that it must be estimable.

- Powers of 2 is also an effective tool to do the estimation.
- **Ideal Days:**
- Another unit of measure. It can be used as a transition for teams that are new to agile.
- It represents an ideal day of work with no interruptions (phone calls, questions, broken builds, etc.)
  However, it doesn't mean an actual day of work to finish.
- Tasks are estimated in hours.
  An estimation is an ideal time (without interruptions/problems).
  Smaller task estimates are more accurate than large.
- After all tasks have been estimated, the hours are totaled up and compared against the remaining hours in the sprint backlog. If there is room, the PBI is added and the team commits to completing the PBI.
  If the new PBI overflows the sprint backlog, the team does not commit and
    - the PBI can be returned to the product backlog and a smaller PBI chosen instead or
    - we can break the original PBI into smaller chunks or
    - we can drop an item already in the backlog to make room or
    - the product owner can help decide the best course of action.
- **Tracking Progress:**
- Information about progress, impediments and sprint backlog of tasks needs to be readily available.
- How close a team is to achieving their goals is also important.
- Scrum employs a number of practices for tracking this information:
    1. Task cards
    2. Burndown charts
    3. Task boards
    4. War rooms (standups)
- **Burndown Charts:**
- Example of a burndown chart:



- Indicates how much work has been done in terms of how many user stories have been completed and when.
- Burndown charts are on Jira.
- On the beginning of the sprint, on the vertical axis, is the number of user stories you'd like to implement.
- At the end of the sprint, the number of user stories should be decreased to 0. That means all the stories have been implemented.
- A burndown chart is a graphical representation of work left to do versus time. It is often used in agile software development methodologies such as Scrum.
- Typically, in a burndown chart, the outstanding work is often on the vertical axis, with time along the horizontal. It is useful for predicting when all of the work will be completed.
- **Taskboard:**
- Example of a taskboard:

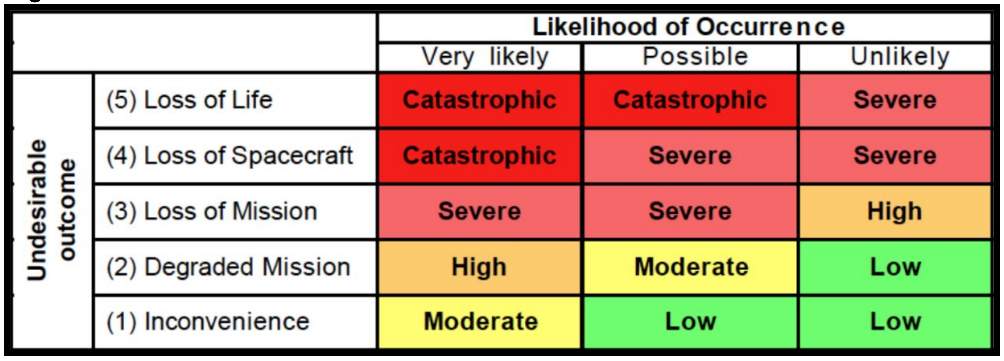


- Both taskcards and taskboards are on Jira.
- The leftmost column are the stories to be implemented and there are 3 columns describing the progress of the tasks.
- In scrum the **task board** is a visual display of the progress of the scrum team during a sprint. It presents a snapshot of the current sprint backlog allowing everyone to see which tasks remain to be started, which are in progress and which are done.
- Simply put, the task board is a physical board on which the user stories which make up the current sprint backlog, along with their constituent tasks, are displayed. Usually this is done with index cards or post-it notes.
- The task board is usually divided into the columns listed below.
  Stories: This column contains a list of all the user stories in the current sprint backlog.
  Not started: This column contains sub tasks of the stories that work has not started on.
  In progress: All the tasks on which work has already begun.
  Done: All the tasks which have been completed.

<u>**Risk Management:**</u>

- **Introduction to Risk:**
- Risk is the possibility of suffering loss.
- Risk itself is not bad; it's essential to progress.
- The challenge is to manage the amount of risk.

- **Risk Exposure (RE):**
- RE = Probability of risk occurring * Total loss if risk occurs
- Calculates the effective current cost of a risk and can be used to prioritize risks that require countermeasures.
  I.e. Can help find which countermeasures work best.
- Higher RE means more serious risk.
- **Risk Reduction Leverage (RRL):**
- RRL = $\frac{Reduction\ in\ Risk\ Exposure}{Cost\ of\ the\ countermeasure}$
- Calculates a value for the return on investment for a countermeasure and can be used to prioritize possible countermeasures.
- Higher RRL indicates more cost-effective countermeasures.
- **Risk Assessment:**
- Quantitative:
  - Measures risk exposure using standard cost and probability measures.
  - **Note:** Probabilities are rarely independent.
- Qualitative:
  - Create a risk exposure matrix.
  - E.g. for NASA



		Likelihood of Occurrence		
		Very likely	Possible	Unlikely
Undesirable outcome	(5) Loss of Life	Catastrophic	Catastrophic	Severe
	(4) Loss of Spacecraft	Catastrophic	Severe	Severe
	(3) Loss of Mission	Severe	Severe	High
	(2) Degraded Mission	High	Moderate	Low
	(1) Inconvenience	Moderate	Low	Low

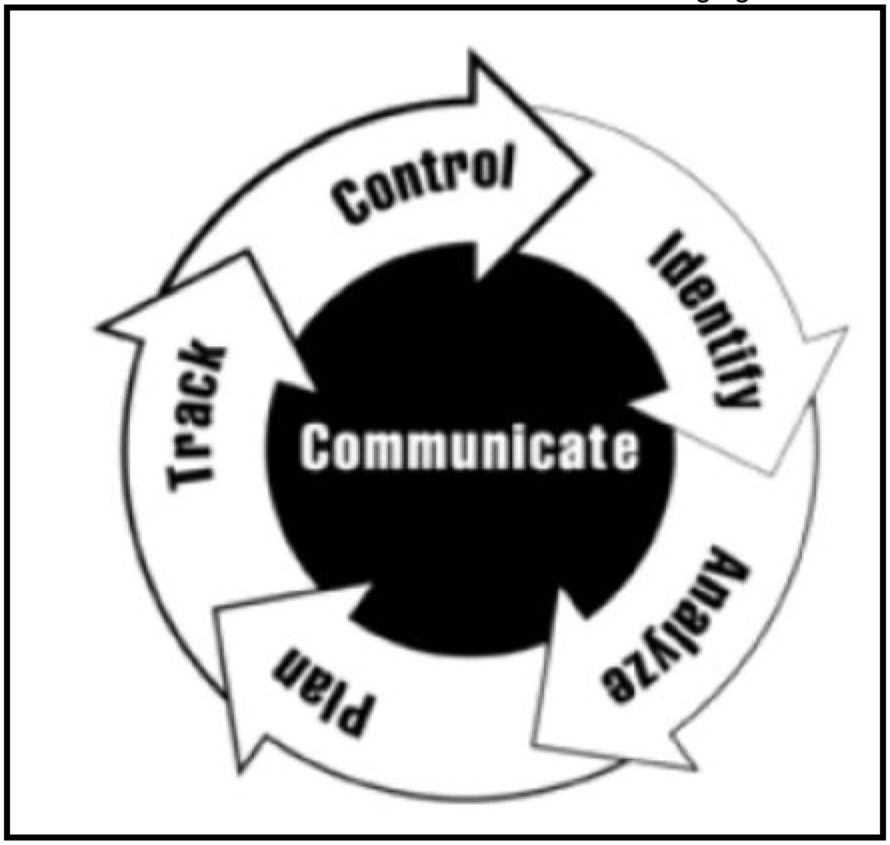
- **Top Software Engineering Risks With Countermeasures:**

Risks	**Countermeasures**
Personnel Shortfalls	Use top talent. Team building. Training.
Unrealistic schedule/budget	Multisource estimation. Designing to cost. Requirements scrubbing.
Developing the wrong software function	Better requirements analysis. Organizational/Operational analysis.
Developing the wrong user interface	Prototypes, scenarios, task analysis.
Gold plating	Requirements scrubbing. Cost benefit analysis. Designing to cost.

Continuing stream of requirement changes	High change threshold. Information hiding. Incremental development.
Shortfalls in externally furnished components	Early benchmarking. Inspections, compatibility analysis.
Shortfalls in externally performed tasks	Pre-award audit. Competitive designs.
Real-time performance shortfalls	Targeted analysis. Simulations, benchmarks, models.
Straining computer science capabilities	Technical analysis. Checking scientific literature.

- **Principles of Risk Management:**
- Global perspective:
    - View software in the context of a larger system.
    - For any opportunity, identify both potential value and potential impacts of adverse results.
- Forward looking view:
    - Anticipate possible outcomes.
    - Identify uncertainty.
    - Manage resources accordingly.
- Open communications:
    - Free-flow information at all project levels.
    - Value the individual voice. Everyone has unique knowledge and insights.
- Integrated management:
    - Project management is risk management.
- Continuous process:
    - Continually identify and manage risks.
    - Maintain constant vigilance.
- Shared product vision:
    - Everyone understands the mission.
    - Focus on results.
- Teamwork:
    - Work cooperatively to achieve the common goal.
- **Continuous Risk Management:**
- Control:
    - Correct for deviations from the risk mitigation plans.
- Identify:
    - Search for and locate risks before they become problems.
    - Use systematic techniques to discover risks.
- Analyze:
    - Transform risk data into decision-making information.
    - For each risk, evaluate:
        - Probability
        - Impact
        - Timeframe
    - Classify and prioritize risks.

- Plan:
    - Choose risk mitigation actions.
- Track:
    - Monitor risk indicators.
    - Reassess risk.
- Communicate:
    - Share information on current and emerging risks.


Control
Identify
Analyze
Plan
Track
Communicate