Why Do Projects Fail:

- Some of the most common reasons projects fail are:
    - Unrealistic or unarticulated project goals.
    - Inaccurate estimates of needed resources.
    - Badly defined system requirements.
    - Poor reporting of the project's status.
    - Unmanaged risks.
    - Poor communication among customers, developers, and users.
    - Use of immature technology.
    - Inability to handle the project's complexity.
    - Sloppy development practices.
    - Poor project management.
- Software project failures have a lot in common with airplane crashes. Just as pilots never intend to crash, software developers don't aim to fail. When a commercial plane crashes, investigators look at many factors, such as the weather, maintenance records, the pilot's disposition and training, and cultural factors within the airline. Similarly, we need to look at the business environment, technical management, project management, and organizational culture to get to the roots of software failures.
- Chief among the business factors are competition and the need to cut costs. Increasingly, senior managers expect IT departments to do more with less and do it faster than before. They view software projects not as investments but as pure costs that must be controlled.
- A lack of upper-management support can also damn an IT undertaking. This runs the gamut from failing to allocate enough money and manpower to not clearly establishing the IT project's relationship to the organization's business.
- Frequently, IT project managers eager to get funded resort to a form of liar's poker, overpromising what their project will do, how much it will cost, and when it will be completed. Many, if not most, software projects start off with budgets that are too small. When that happens, the developers have to make up for the shortfall somehow, typically by trying to increase productivity, reducing the scope of the effort, or taking risky shortcuts in the review and testing phases. These all increase the likelihood of error and, ultimately, failure.
- Sloppy development practices are a rich source of failure, and they can cause errors at any stage of an IT project. To help organizations assess their software-development practices, the U.S. Software Engineering Institute, in Pittsburgh, created the Capability Maturity Model, or CMM. It rates a company's practices against five levels of increasing maturity. Level 1 means the organization is using ad hoc and possibly chaotic development practices. Level 3 means the company has characterized its practices and now understands them. Level 5 means the organization quantitatively understands the variations in the processes and practices it applies.
- Project managers also play a crucial role in software projects and can be a major source of errors that lead to failure. The most important function of the IT project manager is to allocate resources to various activities. Beyond that, the project manager is responsible for project planning and estimation, control, organization, contract management, quality management, risk management, communications, and human resource management. Bad decisions by project managers are probably the single greatest cause of software failures today. Poor technical management, by contrast, can lead to technical errors, but those can generally be isolated and fixed. However, a bad project management decision, such as hiring too few programmers or picking the wrong type of contract, can wreak havoc.

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:

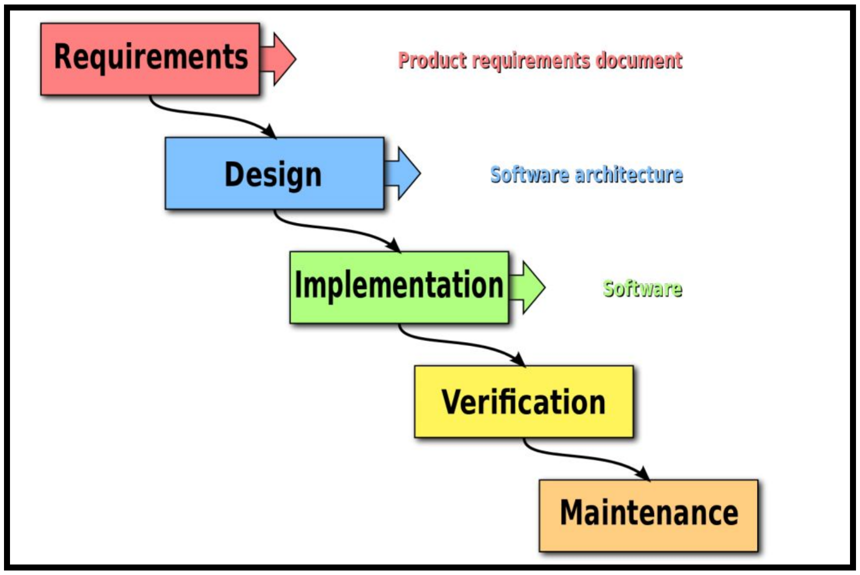
- Software process is a coherent set of activities for specifying, designing, implementing and testing software systems.
- A software process model is an abstract representation of a process that presents a description of a process from some particular perspective. When we describe and discuss about process models, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc and the ordering of these activities.
- Software process descriptions may include:
    - **Products:** The outcomes of a process activity.
    - **Roles:** Reflect the responsibilities of the people involved in the process.
    - **Pre and post-conditions:** Are statements that are true before and after a process activity has been enacted or a product produced.
- 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.

Plan-Driven and Agile Processes:

- **Plan-driven processes** are processes where all of the process activities are planned in advance and progress is measured against this plan.
- In **agile processes**, planning is incremental which makes it easier to change the process to reflect changing customer requirements.
- In practice, most practical processes include elements of both plan-driven and agile approaches.
- There are no right or wrong software processes.

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

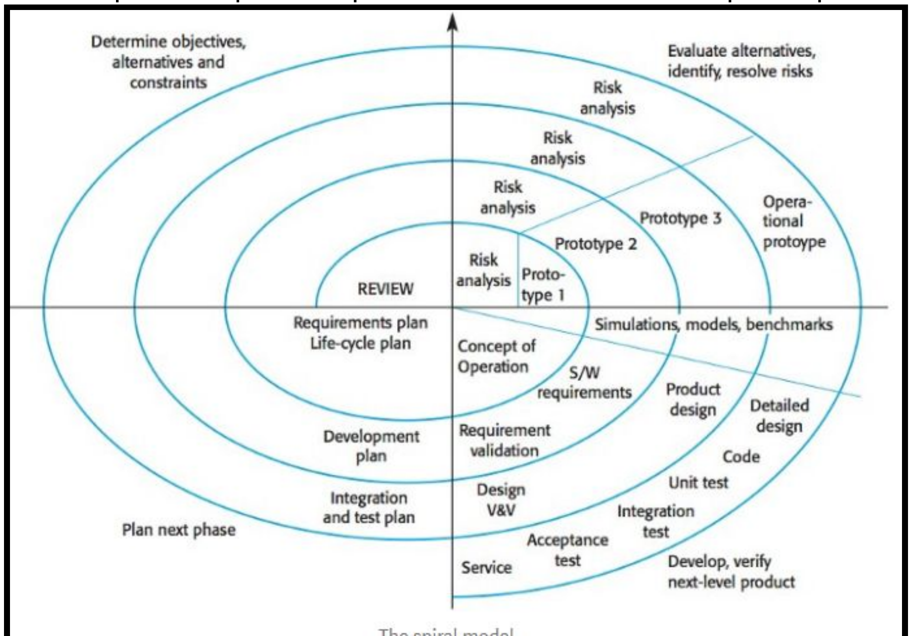


- The result of each phase is one or more documents that should be approved and the next phase shouldn't be started until the previous phase has completely been finished. I.e. Each phase is carried out completely, for all requirements, before proceeding to the next. Furthermore, this process is strictly 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 partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements. Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process. However, few business systems have stable requirements.
    - The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites. In these circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

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.

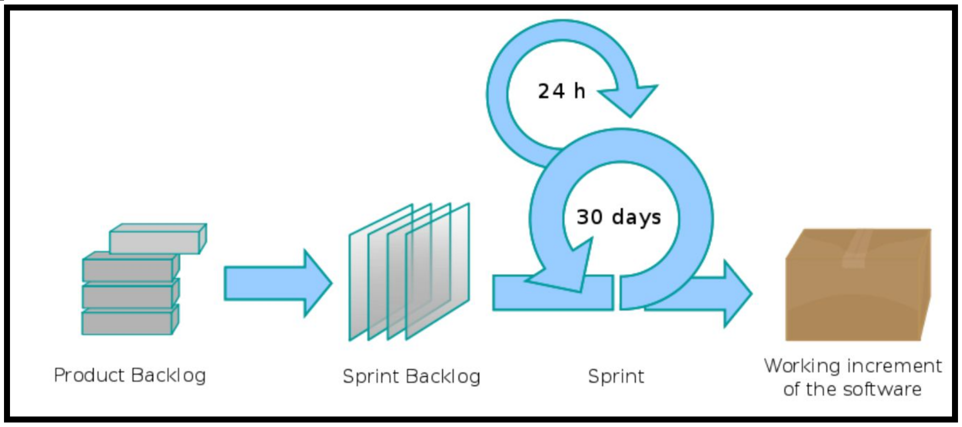
- Advantages of TDD:
    - **Early bug notification:**
        - Using TDD, over time, a suite of automated tests is built up that you and any other developer can rerun at will.
    - **Better designed, cleaner and more extensible code:**
        - TDD helps developers understand how the code will be 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

Why Use Agile:

- **Demand for higher quality with lower cost.**
- **Post-mortems of software projects lead to a lot of knowledge gain about what went right/wrong during the development phase.**
    - With the waterfall model, we do not have a clear way to predict the future, so these lessons are always in hindsight.
    - Smaller, iterative schedules mean problems can be identified/addressed much earlier in the cycle.
- **Agile eliminates waste.**
    - Making changes at the end of a production cycle is costly. With agile we are more likely to detect these changes early enough to reduce the costs.
    - Iterative design means we build a product in small steps.
        - Incrementally add features.
        - Software is in working condition at least every few weeks.
        - Allows people to test earlier in the development cycle.
        - Software improvements happen much earlier and can be fine-tuned rather then trying to modify the design at the end of a waterfall cycle.

Agile Team Management:

- From the bottom up.
- Teams are empowered to manage the smallest level of details, while leaving the higher levels to upper management.
- Teams, upon seeing the small amount of ownership they get from solving smaller problems, take on responsibility for larger problems.
    - Head off issues before they become major problems.
    - Individuals solve problems with their colleagues.

Agile Project Structure:

- An agile project consists of a series of iterations of development.
- Each interval usually lasts only two to four weeks.
- Developers implement features, called user stories, during each iteration that add value to the project.
- Each iteration contains a full development cycle:
    - Concept
    - Design
    - Coding
    - Testing
    - Deployment
- The project is reviewed at the end of each iteration.
- Results are used to direct future iterations.
- Every three to six iterations the project is built up to a release state, meaning that most major goals are accomplished.

Incremental Development Benefits:

- The cost of accommodating changing customer requirements is reduced as the amount of analysis and documentation that has to be redone is much less than what is required with the waterfall model.
- It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented.
- More rapid delivery and deployment of useful software to the customer is possible. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental Development Problems:

- The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- The system structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

Is It True That Only Non-Agile Projects Fail:

- The Scott Ambler survey defines success as a solution being delivered and meeting its success criteria within the acceptable range defined by the organization, and failure as the project never delivering a solution.
- The Ambler report concluded that agile projects do not fail more than other projects. They succeed at the same level as other iterative methodologies.
- However, agile projects face a set of challenges and problems related to applying a different approach to project management. According to VersionOne, the top three reasons for agile project failure are:
    1. Inadequate experience with agile methods.
    2. Little understanding of the required broader organizational change.
    3. Company philosophy or culture at odds with agile values.

Selecting a Development Model:

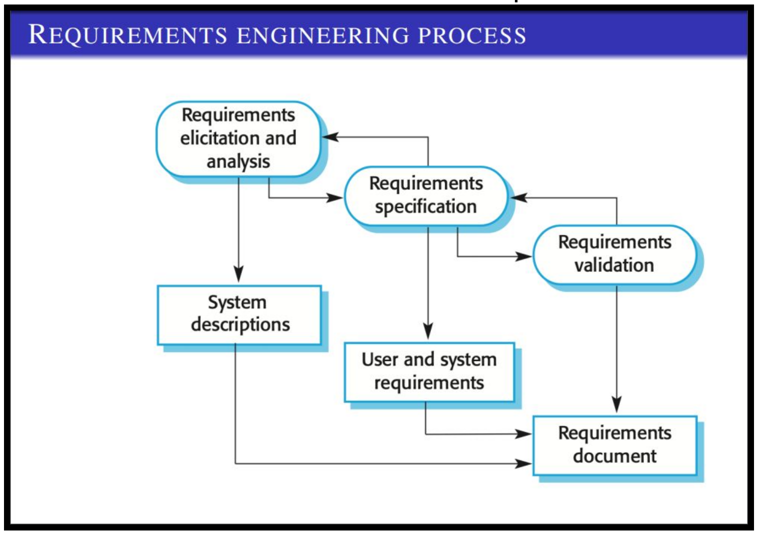
- For organizations and projects, where experience can be used to plan a course of action with a good degree of certainty for a positive outcome, a traditional methodology may be more appropriate than an agile methodology. In this case, the plans can be developed up-front and then designed, developed, and tested without much variance.
- Agile methodologies are effective when the product details cannot be defined or agreed in advance with any degree of accuracy. This situation calls for the collaborative environment between the user and the developer. Agile methodologies are suited for a dynamic and changing environment.

Key Process Stages:

- The stages are:
    1. Requirements specification
    2. Software discovery and evaluation
    3. Requirements refinement
    4. Application system configuration
    5. Component adaptation and integration
- Real software processes are interleaved sequences of technical, collaborative and managerial activities with the overall goal of specifying, designing, implementing and testing a software system.
- The four basic process activities of specification, development, validation and evolution are organized differently in different development processes. For example, in the waterfall model, they are organized in sequence, whereas in the incremental development they are interleaved.

Software Specification:

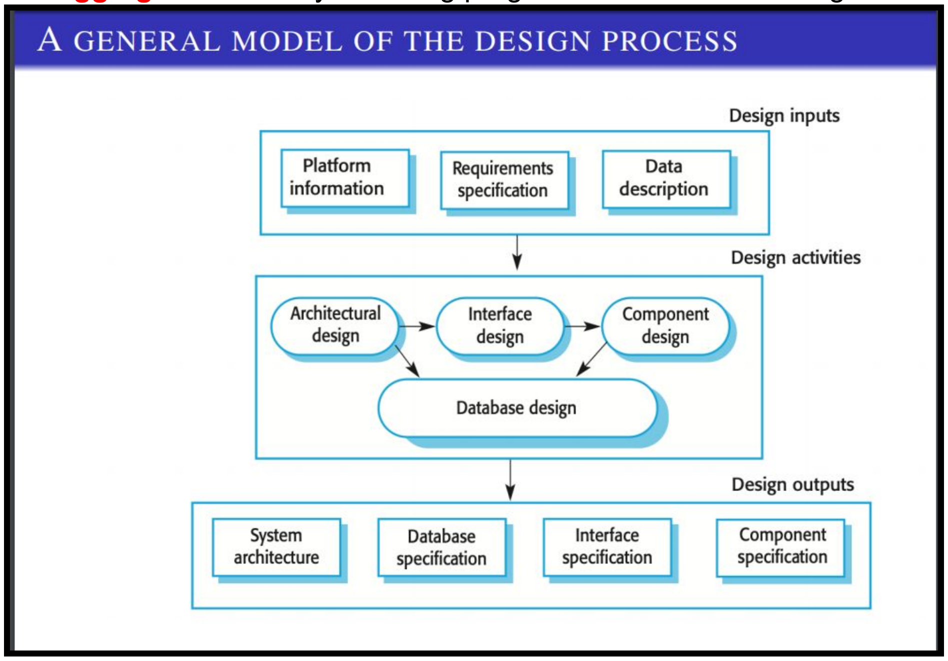
- Is the process of establishing what services are required and the constraints on the system's operation and development.
- **Requirements engineering process** involves the following:
  - **Requirements elicitation and analysis:** Is the practice of researching and discovering the requirements of a system from users, customers, and other stakeholders. It is related to the various ways used to gain knowledge about the project domain and requirements. The various sources of domain knowledge include customers, business manuals, the existing software of the same type, standards and other stakeholders of the project.It answers the question "What do the system stakeholders require or expect from the system?"
  - **Requirements specification:** Defines the requirements in detail. This activity is used to produce formal software requirement models. During specification, more knowledge about the problem may be required which can again trigger the elicitation process.
    I.e. It is the process of writing down the **user and system requirements** into a document. The requirements should be clear, easy to understand, complete and consistent.
    The **user requirements** for a system should describe the functional and non-functional requirements so that they are understandable by users who don't have technical knowledge. You should write user requirements in natural language supplied by simple tables, forms, and intuitive diagrams.
    The requirement document shouldn't include details of the system design and you shouldn't use any software jargon or formal notations.
    The **system requirements** are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They shouldn't be concerned with how the system should be implemented or designed. The system requirements may also be written in natural language but other ways based on structured forms, or graphical notations are usually used.
  - **Requirements validation:** Checks the validity of the requirements. It's a process of ensuring the specified requirements meet the customer needs.
    I.e. It refers to a different set of tasks that ensures that the software that has been built is traceable to customer requirements.


REQUIREMENTS ENGINEERING PROCESS
Requirements elicitation and analysis
Requirements specification
Requirements validation
System descriptions
User and system requirements
Requirements document

Software Design and Implementation:

- Is the process of converting the system specification into an executable system.
- **Software design** is designing a software structure that realizes the specification.
  Some software design activities include:
  - **Architectural design:** Identifying and defining the overall structure of the system, the principal components, their relationships and how they are distributed.
  - **Database design:** Designing the system data structures and how they will be represented in a database.
  - **Interface design:** Defining the interfaces between system components. The interface specification must be clear. Therefore, a component can be used without having to know it's implemented. Once the interface specifications are agreed, the components can be designed and developed concurrently.
  - **Component selection and design:** Searching for reusable components. If unavailable, you design how it will operate.

  I.e. A software design is a description of the structure of the software to be implemented, data models, interfaces between system components, and maybe the algorithms used.
- **Software implementation** is taking your design and translating into an executable program.
  I.e. The implementation phase is the process of converting a system specification into an executable system. If an incremental approach is used, it may also involve refinement of the software specification.
  The software is implemented either by developing program(s) or by configuring an application system.
  Design and implementation are interleaved activities for most types of software system.
  Programming is an individual activity with no standard process.
  Debugging is the activity of finding program faults and correcting these faults.



- The activities of design and implementation are closely related and may be interleaved.

Software Validation:

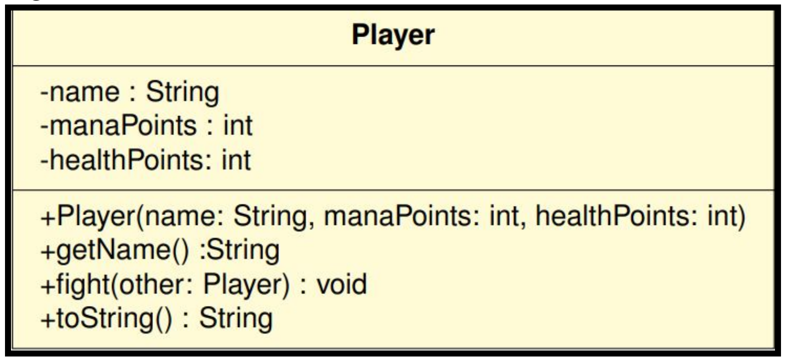
- **Verification and validation (V&V)** is intended to show that a system conforms to its specification and meets the requirements of the system customer.
  I.e. It is the process of checking that a software system meets specifications and that it fulfills its intended purpose.
- **Software validation** is a dynamic mechanism of testing and validating if the software product actually meets the exact needs of the customer or not. The process helps to ensure that the software fulfills the desired use in an appropriate environment. The validation process involves activities like unit testing, integration testing, system testing and user acceptance testing.
  I.e. Software validation checks if the code actually does what it is supposed to do.
- **Software verification** is a process of checking documents, design, code, and program in order to check if the software has been built according to the requirements or not. The main goal of the verification process is to ensure quality of software application, design, architecture etc. The verification process involves activities like reviews, walk-throughs and inspection.
  I.e. Software verification checks if the program is built according to the specifications and design.
- Involves checking and review processes and system testing.
- **System testing** involves executing the system with test cases that are derived from the specification of the real data to be processed by the system. The purpose of this test is to evaluate the system's compliance with the specified requirements.
  I.e. System testing is the process of testing an integrated system to verify that it meets specified requirements.
- Testing is the most commonly used V&V activity.

Unified Modeling Language (UML):

- **Unified Modeling Language (UML)** allows us to express the design of a program before writing any code.
- It is language-independent.
- It is an extremely expressive language.
- UML is a graphical language for visualizing, specifying, constructing, and documenting information about software-intensive systems.
- UML can be used to develop diagrams and provide programmers with ready-to-use, expressive modeling examples. Some UML tools can generate program language code from UML. UML can be used for modeling a system independent of a platform language.
- UML is a picture of an object oriented system. Programming languages are not abstract enough for object oriented design. UML is an open standard and lots of companies use it.
- Legal UML is both a descriptive language and a prescriptive language. It is a descriptive language because it has a rigid formal syntax, like programming languages, and it is a prescriptive language because it is shaped by usage and convention.
- It's okay to omit things from UML diagrams if they aren't needed by the team/supervisor/instructor.
- **UML diagrams can help engineering teams:**
  - Bring new team members or developers switching teams up to speed quickly.
  - Navigate source code.
  - Plan out new features before any programming takes place.
  - Communicate with technical and non-technical audiences more easily.

- **Uses of UML:**
    1. It can be used as a sketch to communicate aspects of the system.
    - **Forward design:** Doing UML before coding.
    - **Backward design:** Doing UML after coding as documentation.
    2. It can be used as a blueprint to show a complete design that needs to be implemented.
    3. It can be used as a programming language.
    - Some UML tools can generate program language code from UML.
- **Class Diagram:**
    - A **class diagram** describes the structure of an object oriented system by showing the classes in that system and the relationships between the classes. A class diagram also shows constraints, and attributes of classes.
      I.e.
      A UML class diagram is a picture of:
        - The classes in an object oriented system.
        - Their fields and methods.
        - Connections between the classes that interact or inherit from each other.
    - Some things that are not represented in a UML class diagram are:
        - Details of how the classes interact with each other.
        - Algorithmic details, like how a particular behavior is implemented.
    - Each class is represented by a box divided in three sections:
        1. **Class name**
        2. **Data members/Attributes**
        - The data members section of a class lists each of the class's data members on a separate line.
        - Each line uses this format: **attributeName : type**
          E.g. name : String
        3. **Methods/Operations**
        - The methods of a class are displayed in a list format, with each method on its own line.
        - Each line uses this format:
          methodName(param1: type1, param2: type2, ...) : returnType
          E.g. fight(other: Player) : void

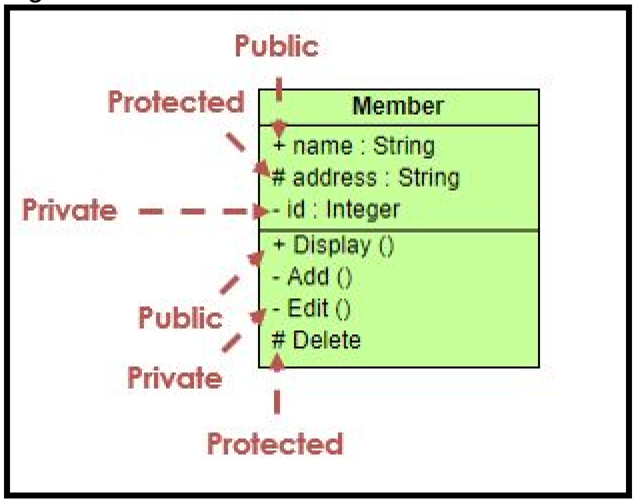
      E.g.



```
Player
-name : String
-manaPoints : int
-healthPoints: int
+Player(name: String, manaPoints: int, healthPoints: int)
+getName() :String
+fight(other: Player) : void
+toString() : String
```

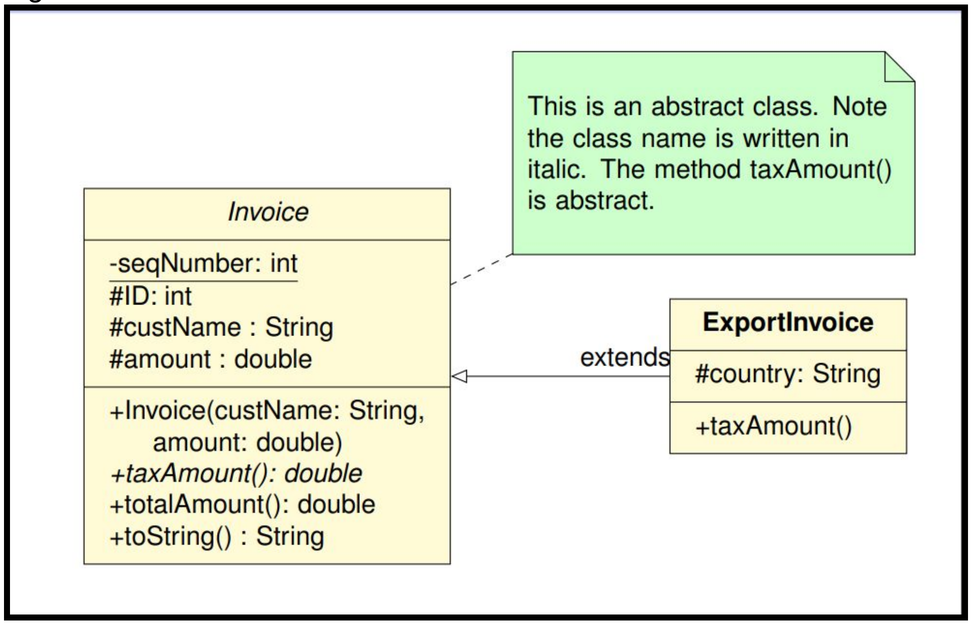
- Visibility:
    - − means that it is private.
    - + means that it is public.
    - # means that it is protected.
    - ~ means that it is a package.

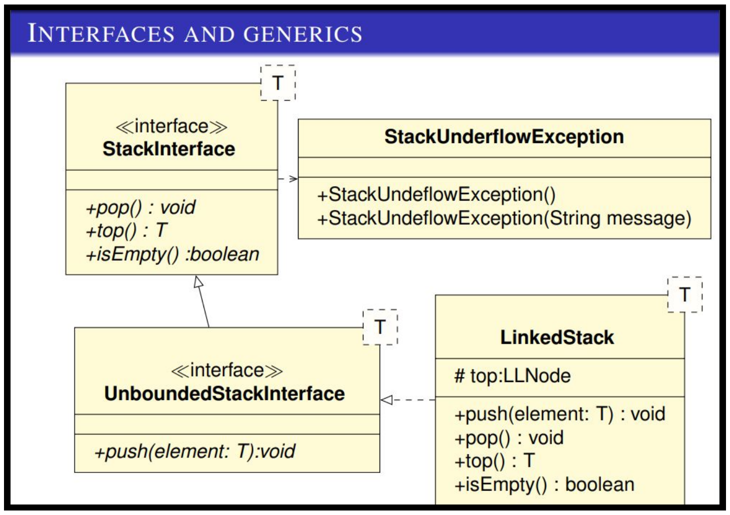
  E.g.



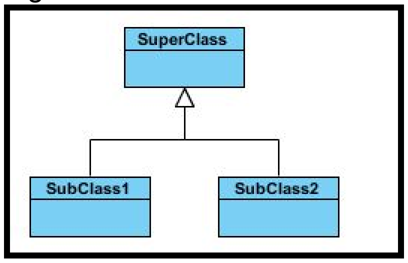
- Notation:
    - To show something is static, underline it.
    - To show that a method is abstract, italicize the method name.
    - To show that a class is abstract, either italicize the class name or do <<abstract>>.
    - To show that something is an interface, do <<interface>>.

  E.g.

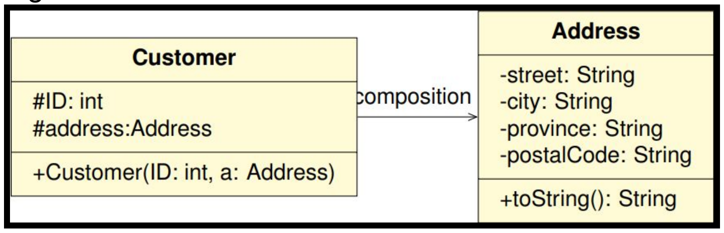




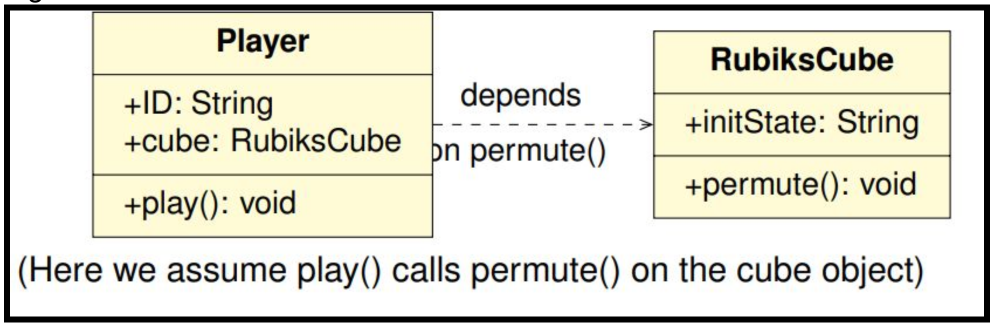
- Inheritance:
    - Inheritance is represented by an arrow with a hollow arrow head that points to the parent class. The other end of the arrow is the child class.
    - E.g.



- Composition:
    - It has the following characteristics:
        1. It is a binary association.
        2. It is a whole/part relationship. The whole is also called the **composite**. The composite is made up of many other objects/parts.
        3. The composite "has a" part. (Has-a relationship)
           E.g. A car has an engine.
           Car would be the composite.
           Engine would be the part.
        4. A part could be included in at most one composite at a time, but a composite can have multiple parts.
        5. If a composite is deleted, all of its composite parts are deleted with it.
    - With composition, we draw an arrow such that the arrow points to the parts class.
    - E.g.



- Dependency:
    - Dependency indicates a “uses” relationship between two classes.
    - If class A “uses” class B, then one or more of the following statements generally hold true:
        1. Class B is used as the type of a local variable in one or more methods of class A.
        2. Class B is used as the type of parameter for one or more methods of class A.
        3. Class B is used as the return type for one or more methods of class A.
        4. One or more methods of class A invoke one or more methods of class B.
    - Dependency is represented by a dotted arrow. The arrow head points to the independent element and the other end represents the dependent element.
    - E.g.


Player
+ID: String
+cube: RubiksCube
+play(): void
depends
on permute()
RubiksCube
+initState: String
+permute(): void
(Here we assume play() calls permute() on the cube object)