* What is Software Engineering?
* “An organized, analytical approach to the design, development, use, and maintenance of software.”
* Software Engineering is the application of systematic,disciplined,quantifiable approach to the development, operation and maintenance of software.
* Advantage of being a software
* In software we can make changes to any part of a program even after it is completely written.
* We can refine the program during development to better meet user needs.
* We can add new features to take advantage of opportunities discovered during implementation, and make modifications to meet evolving business needs.
* Why is Software Engineering Important?
* Software engineering includes techniques for avoiding the many pitfalls that otherwise may lead to failure.
* It ensures the final application is effective, usable, and maintainable.
* It helps you meet milestones on schedule and produce a finished project on time and within budget.
* Software engineering gives you the flexibility to make changes to meet unexpected demands.
* Requirement Gathering
* Requirements are the features that your application must provide.
* At the beginning of the project, you gather requirements from the customers to figure out what you need to build.
* Throughout development, you use the requirements to guide development and ensure that you’re heading in the right direction
* The properties of useful requirement.
* **Clear**- Good requirements are clear, concise, and easy to understand.

To be clear, requirements cannot be vague or ill‐defined.

* **Unambiguous**- In addition to being clear and concrete, a requirement must be unambiguous.
* **Consistent-**  A project’s requirements must be consistent with each other. That means not only that they cannot contradict each other, but that they also don’t provide so many constraints that the problem is unsolvable.
* **Prioritized**. You might like to include every feature but don’t have the time or budget, so something’s got to go. At this point, you need to prioritize requirements.
* **Verifiable** Requirements must be verifiable. If you can’t verify a requirement, how do you know whether you’ve met it? Being verifiable means the requirements must be limited and precisely defined.
* **Words to Avoid** Some words are ambiguous or subjective, and adding them to a requirement can make the whole thing fuzzy and imprecise.
* **Comparatives** —Words like faster, better, more, and shinier.
* **Imprecise adjectives** —Words like fast, robust, user‐friendly, efficient, flexible, and glorious.
* **Vague commands** —Words like minimize, maximize, improve, and optimize
* Requirement Categories

**Audience Oriented Requirements**

* Focus on different audiences and the different point of view that each audience has.
* They use a somewhat business‐oriented perspective to classify requirements according to the people who care the most about them.

.

* Business Requirements

Business Requirements

Layout project’s high level goals.

* User Requirements
* Functional Requirements
* Non Functional Requirements
* Implementation Requirements

* Business Requirements
* **User Requirements**

Describe how project will be used by end-users.

* **Functional Requirements**

Detailed statements of the project’s desired capabilities.

Example: The reports that the application produces, interfaces to other applications .

* Business Requirements
* **Non Functional Requirements**
* Statements about the quality of the application’s behavior.
* Applications performance , reliability and security characteristics.
* Business Requirements
* **Implementation Requirements**
* Temporary features that are needed to transition to using the new system but that will be later discarded.
* The task described in implementation requirement don’t always involve programming.
* Examples such as hiring new staff, buying new hardware, preparing training materials, Training users to use the system .

* System Requirement Categories(FURPS)
* **Functionality** —What the application should do. These requirements describe the system’s general features including what it does, interfaces with other systems, security etc.
* **Usability** —What the program should look like. These requirements describe self‐oriented features such as the application’s general appearance, ease of use, navigation methods, and responsiveness.
* System Requirement Categories
* **Reliability** —How reliable the system should be. These requirements indicate such things as when the system should be available ,how often it can fail, and how accurate the system is.
* **Performance** —How efficient the system should be. These requirements describe such things as the application’s speed, memory usage, disk usage, and database capacity.
* System Requirement Categories
* **Supportability** —How easy it is to support the application. These requirements include such things as how easy it will be to maintain the application, how easy it is to test the code, and how flexible the application is.
* FURPS+

FURPS was extended into FURPS+ to add a few requirements categories that software engineers thought were missing.

* **Design constraints** —These are constraints on the design that are driven by other factors such as the hardware platform, software platform, network characteristics, or database.
* **Implementation requirements** —These are constraints on the way the software is built.
* FURPS+
* **Interface requirements** —These are constraints on the system’s interfaces with other systems. They tell what other systems will exchange data with the one you’re building.
* **Physical requirements** —These are constraints on the hardware and physical devices that the system will use.
* GATHERING REQUIREMENTS

**Listen to Customers (and Users)**

* Learn as much as you can about the problem they are trying to address and any ideas they may have about how the application might solve that problem.
* Initially, focus as much as possible on the problem, not on the customers’ suggested solutions, so you can keep the requirements flexible.
* If the customers insist on a particular feature that you think is unimportant, or if they request something that just seems strange, ask them why they want it.
* Use the Five Ws (and One H)
* Who
* Ask who will be using the software and get to know as much as you can about those people.

* Find out if the users and the customers are the same and learn as much about the users as you can.
* What
* Figure out what the customers need the application to do.
* Focus on the goals as much as possible rather than the customers’ ideas about how the solution should work.
* When
* Find out when the application is needed. If the application will be rolled out in phases, find out

which features are needed when.

* Where
* Find out where the application will be used. Will it be used on desktop computers in an air-conditioned

office? Or will it be used on phones in a noisy subway?

* Why
* Ask why the customers need the application. Find out if there is a real reason to believe a new application will help.
* How
* If the users are used to doing something a certain way, you may reduce training time
* Refining Requirements
* You should have a good understanding about the users’ current operations and needs.
* You need to distill the goals (what the customers need to do) into approaches (how the application will do it).
* Three approaches for converting goals into requirements.
  + **Copy Existing Systems**
  + **Clairvoyance**
  + **Brainstorm**
* Copy Existing Systems

* If you’re building a system to replace an existing system or a manual process, you can often use many of the behaviors of the existing system as requirements for the new one.

**Advantages**

* It doesn’t take an enormous amount of software engineering experience to dig through an existing application and write down what it does.
* This approach also makes it more likely that the requirements can actually be satisfied, at least to the extent the current system works.
* This approach provides an unambiguous example of what you need to do.

**Disadvantages**

* You probably wouldn’t be building a new version of an existing system unless you planned to make some changes.
* Users are often reluctant to give up even the tiniest features in an existing program.
* Clairvoyance
* This technique is particularly effective if the project team has previously built a similar system.
* In this case, the team already knows more or less what the application needs to do, which things will be easy and which will be hard, how much time everything requires.

**Advantages**

* Having an experienced project lead greatly increases the chances that the requirements will include everything you need to make the project succeed.
* It also greatly increases the chances that the team will anticipate problems and handle them easily as development continues.
* Documenters who have written user manuals for similar applications will find writing manuals for the new project easier.
* Project managers with similar experience will know what tasks are likely to be difficult.
* Even customers with previous software engineering experience will be better at creating good requirements.

**Disadvantages**

* This technique will not lead you to new innovative solutions that might be better than the old ones
* Brainstorm
* Brainstorming helps to create more creative solutions.
* The basic approach that most people think of as brainstorming is called the *Osborn method developed by* Alex Faickney Osborn.
* The gist of the method is:
* To gather as many ideas as possible, not worrying about their quality or practicality.
* After you assemble a large list of possible ideas, you examine them more closely to see which deserve further work.
* Try to get a diverse group of participants which include customers, users, user interface designers, system architects, team leads, programmers, trainers, and anyone else who has an interest in the project.
* Get as many different viewpoints as you can.
* To keep the ideas flowing, don’t judge or critique any of the ideas. If you criticize someone’s ideas, that person may shut down and stop contributing.
* Write down every idea no matter how impractical it may seem
* Osborn’s method

Osborn’s method uses the following four rules:

1. **Focus on quantity.** Do everything you can to keep the ideas flowing. The more ideas you collect, the greater your chances of finding a really creative and revolutionary solution.

2. **Withhold criticism.** Criticism can make people stop contributing.

* **3.Encourage unusual ideas** . You can always “tone down a wild idea” but you may need to think way outside of the box to find really creative solutions.
* 4. **Combine and improve ideas.** Form new ideas by combining other ideas or using one idea to modify another.
* RECORDING REQUIREMENTS
* After you decide what should be in the requirements, you need to write them down so that everyone can read them.
* Methods for recording requirements are:
* **UML**
* **User Stories**
* **Use Cases**
* **Prototypes**

**UML**

* The Unified Modeling Language (UML) lets you specify how parts of the system should work.
* It uses several kinds of diagrams to represent

different pieces of the system

**User Stories**

* A *user story* is exactly what you might think: a short story explaining how the system will let the user do something.

**Advantage**

* They are easy to write, easy to understand, and can cover just about any situation you can imagine.
* Your customers, developers, managers, and other team members already know how to understand stories without any new training.

**Disadvantage of User Stories**

* You can easily write stories that are confusing, ambiguous, inconsistent with other stories, and unverifiable.

**Use Cases**

* A *use case* is a description of a series of interactions between actors. The actors can be users or parts of the application.

Template of an use case

* **Title** —the title includes an action (examines) and the main actor (user).
* **Main success scenario** —A numbered sequence of steps describing the most normal variations of the scenario.
* **Extensions** —Sequences of steps describing other variations of the scenario. This may include cases such as when the user enters invalid data or the application can’t handle a request.

**Prototypes**

* A *prototype* is a mockup of some or all of the application.
* The idea is to give the customers a feel for what the finished application will look like and how it will behave than you can get from text descriptions such as user stories and use cases.
* A simple user interface prototype might display forms that contain labels, text boxes, and buttons showing what the finished application will look like.
* In *nonfunctional prototype*, the buttons, menus, and other controls on the forms wouldn’t actually do anything.
* A *functional prototype* (or *working prototype*) looks and acts as much like the finished application.
* It might use less efficient algorithms.
* Load data from a text file instead of a database, or display random messages instead of getting them from another system. It might even use hard‐coded fake data.

Two different kinds of prototype are :

* Throwaway Prototype
* Evolutionary Prototype

**Throwaway Prototype :**

* After you’ve fine‐tuned the prototype so that it represents the customers’ requirements , you can leave it alone.
* You can continue to refer to it if there’s a question about what the application should look like or how it should work.
* But you start over from scratch when building the application. This kind of prototype is called a *throwaway prototype*.

**Evolutionaryprototype:**

* You can start replacing the prototype code and fake data with production‐quality code and real data.
* Over time, you can evolve the prototype into increasingly functional versions until eventually it becomes the finished application. This kind of prototype is sometimes called an *evolutionary prototype*.
* Requirement Specification
* VALIDATION AND VERIFICATION
* *Requirement validation* is the process of making sure that the requirements say the right things.
* Someone, often the customers or users, need to work through all the requirements and make sure that

they: (1) Describe things the application should do.

(2) Describe *everything* the application should do.

* *Requirement verification* is the process of checking that the finished application actually satisfies the requirements.
* High Level Design
* High‐level design provides a view of the system at an abstract level.
* It shows how the major pieces of the finished application will fit together and interact with each other.
* High Level Design is the overall system design covering the system architecture.
* It describes the relation between various modules and function of the system.

**Some of the most common items you might want to specify in the high‐level design**.

* **Security**
* **Hardware**
* **User Interface**
* **Architecture**
* **Reports**
* **Other Outputs**
* **Database**
* **Configuration Data**
* **Data flows and States**
* **Training**

**Security**

The high‐level design should sketch out all the application’s security needs.

* **Operating system security—**This includes the type of login procedures, password expiration policies, and password standards.
* **Application security—**A separate application username and password. Application security also means providing the right level of access to different users.
* **Data security—** Protecting data from destructive forces and unwanted actions of unauthorized users.
* **Network security—**Network security consists of policies and practices adopted to prevent and monitor unauthorized access of computer network and network accessible resources.
* **Physical security—**Many software engineers overlook physical security. Physical security prevents and discourages attackers by installing alarms, cameras etc.

**Hardware**

You can build systems to run on mainframes, desktops, laptops, tablets, and phones.

Additional hardware that you need to specify might include the following:

* Printers
* Network components (cables, modems, gateways, and routers)
* Servers (database servers, web servers, and application servers)
* Specialized instruments (scales, microscopes, programmable signs, and GPS units)
* Audio and video hardware (webcams, headsets, and VOIP)

**User Interface**

* Indicates the main methods for navigating through the application.
* In addition to the application’s basic navigational style, the high‐level user interface design can describe special features such as clickable maps, important tables, or methods for specifying system settings (such as sliders, scrollbars, or text boxes).
* It can also address general appearance issues such as color schemes, company logo placement, and form skins.

**Internal Interfaces**

* Specifies the internal interactions or specify how the pieces will interact.
* The teams assigned to the pieces can work separately without needing constant coordination.
* Specifies these internal interactions clearly and unambiguously so that the teams can work as independently as possible.

**External Interfaces**

* Many applications must interact with external systems.
* External interface are easier to specify than internal interface.

**Architecture**

* An application’s architecture describes how its pieces fit together at a high level.

**Monolithic**

* In a *monolithic architecture* , a single program does everything.
* It displays the user interface, accesses data, processes customer orders, prints invoices, launches missiles, and does whatever else the application needs to do.

**Drawbacks of Monolithic Architecture.**

* The pieces of the system are tied closely together, so it doesn’t give you a lot of flexibility.
* If you get any of the details wrong, the tight coupling between the pieces of the system makes fixing them later difficult.

**Advantages**

* Because everything is built into a single program, there’s no need for complicated communication across networks.
* You don’t need to worry about the network going

down; and you don’t need to worry about network security.

**Client/Server**

* A *client/server architecture* separates pieces of the system that need to use a particular function (clients) from parts of the system that provide those functions

(servers).

* For example, many applications rely on a database to hold information about customers, products, orders, and employees. The application needs to display that information in some sort of user interface. One way to do that would be to integrate the database directly into the application
* One problem with this design is that multiple users cannot use the same data. You can fix that problem by moving to a ***two‐tier architecture*** where a client (the user interface) is separated from the server.

***Two‐tier architecture***

* The clients and server communicate through some network such as a local area network (LAN), wide area

network (WAN), or the Internet.

* The two‐tier architecture support multiple clients with the same server, but it ties clients and servers relatively closely together.

***Three-tier***

* Three-tier helps to increase the separation between the clients and server by introducing a middle-tier.
* If you need to change the way the server stores data, you need to update only the middle tier so that it translates the new format into the version expected by the client.
* If the client’s data needs change, you can modify the middle tier.

**Service‐Oriented Architecture**

* A *service‐oriented architecture* ( *SOA* ) is similar to a component‐based architecture except the pieces are implemented as services.
* A *service* is a self‐contained program that runs on its own and provides some kind of service for its clients.

**Data‐centric or database‐centric architectures**

* Storing data in a relational database system
* Using tables instead of hard‐wired code to control the application
* Using stored procedures inside the database to perform calculations and implement business logic.

**Event‐Driven**

* In an *event‐driven architecture* ( *EDA* ), various parts of the system respond to events as they occur.

**Rule‐Based**

* A *rule‐based architecture* uses a collection of rules to decide what to do next.
* These systems are sometimes called *expert systems* or *knowledge‐based systems* .

Eg: Troubleshooting system

**Distributed**

* In a *distributed architecture* , different parts of the application run on different processors and may run at the same time.
* The processors could be on different computers scattered across the network, or they could be different cores on a single computer.
* Service‐oriented and multitier architectures are often distributed, with different parts of the system running on different computers.

**Mix and Match**

* An application doesn’t need to stick with a single architecture. Different pieces of the application might use different design approaches.

**Reports**

software project can use some kinds of reports.

* Business applications might include reports that deal with customers (who’s buying, who has unpaid bills, where customers live), products (inventory, pricing, what’s selling well), and users (which employees are selling a lot, employee work schedules).

**Other Outputs**

* In addition to normal reports, you should consider other kinds of outputs that the application might create.
* The application could generate printouts (of reports and other things), web pages, data files, image files, audio, video, output to special devices, e‐mail, or text messages.

**Database**

* You need to specify whether the application will store data in text files, XML files, a full‐fledged relational database, or something more exotic such as a temporal database or object store.
* Three common database‐specific issues that you should address during high level design:
* **Audit trails,**
* **User access,**
* **Database maintenance**.

**Audit trails**

* An *audit trail* keeps track of each user who modifies (and in some applications *l* views) a specific record.
* Auditing can be as simple as creating a history table that records a user’s name, a link to the record that was modified and the date when the change occurred.

**User Access**

* Many applications also need to provide different levels of access to different kinds of data.
* One way to handle user access is to build a table listing the users and the privileges they should be given.
* The program can then disable or remove the buttons and menu items that a particular user shouldn’t be allowed to use.

**Database Maintenance**

* Over time database gets disorganized and full of random junk, you need to reorganize so that you can find things efficiently.
* Move the older data into a *data warehouse* , a secondary database that holds older data for analysis.
* Removing old data from a database can help keep it responsive, but a lot of changes to the data can make the database’s indexes inefficient and that can hurt performance. For that reason, we need to periodically run database tuning software to restore peak performance.
* You should design a database backup and recovery scheme.

**Data flows and States**

* In order to describe the system and the way processes interact with the data, many data flows and state transition diagrams are used.

**Training**

* Deciding whether users want to attend courses taught by instructors, read printed manuals, watch instructional videos, or browse documentation online.
* **PREDICTIVE AND ADAPTIVE MODELS**
* **PREDICTIVE MODEL**
* **Predictive development model Predict in advance what needs to be done and then do it.**
* **It uses the requirements to design the system, and use the design as a blueprint to write the code and then test the code.**
* **ADAPTIVE MODEL**
* **An adaptive development model enables you to change the project’s goals if necessary during development.**
* **An adaptive model lets you periodically reevaluate and decide whether you need to change direction.**
* **The adaptive model just gives you chances to fine‐tune the project if necessary.**
* **WATERFALL MODEL**
* ***Waterfall* is the plain vanilla of the predictive model world.**
* **It assumes that you finish each step completely and thoroughly before you move on to the next step.**
* **The waterfall model can work reasonably well if all the following assumptions are satisfied:**
* **The requirements are precisely known in advance.**
* **The requirements include no unresolved high‐risk items.**
* **The requirements won’t change much during development.**
* **The team has previous experience with similar projects so that they know what’s involved in building the application.**
* **There’s enough time to do everything sequentially.**
* **WATERFALL WITH FEEDBACK**
* **(figure)**
* **The *waterfall with feedback* variation enables you to move backward from one step to the previous step.**
* **If you’re working on design and you discover that there was a problem in the requirements, you can briefly go back to the requirements and fix them.**
* **The farther you have to go back up the cascade, the harder it is.**
* **For example, if you’re working on implementation and discover a problem in the requirements, it’s hard to skip back up two steps to fix the problem.**
* **It’s also less meaningful to move back up the cascade for the later steps.**
* **For example, if you find a problem during maintenance, then you should probably treat it as a maintenance task instead of moving back into the deployment stage.**
* **SASHIMI**
* ***Sashimi* (also called *sashimi waterfall* and *waterfall with overlapping phases*) is similar to the waterfall model except the steps are allowed to overlap.**
* **In a project’s first phase, some requirements will be defined while we are still working on others. At that point, some of the team members can start designing the defined features while others continue working on the remaining requirements**
* **A bit later, the design for some parts of the application will be more or less finished but the design for other parts of the system won’t be.**
* **At that point, some developers can start writing code for the designed parts of the system while others continue working on the rest of the design tasks and possibly even on remaining requirements.**
* **ADVANTAGES OF SASHIMI**
* **People with different skills can focus on their specialties without waiting for others.**
* **A second advantage is that it lets you perform a *spike* or *deep dive* into a particular topic to learn more about it.**
* **A third advantage to overlapping phases is it lets later phases modify earlier phases. If you discover during design that the requirements are impossible or need alterations, you can make the necessary changes.**
* **INCREMENTAL WATERFALL**
* **The *incremental waterfall l* model (also called the *multi‐waterfall* model) uses a series of separate waterfall cascades.**
* **Each cascade ends with the delivery of a usable application called an *increment* .**
* **Each increment includes more features than the previous one, so you’re building the final application incrementally.**
* **During each increment, you’ll get a better understanding of what the final application should look like.**
* **V‐MODEL**
* ***V‐model* is basically a waterfall that’s been bent into a V shape.**
* **The tasks on the left side of the V break the application down from its highest conceptual level into more and more detailed tasks.**
* **This process of breaking the application down into pieces that you can implement is called *decomposition***
* **The tasks on the right side of the V consider the finished application at greater and greater levels of abstraction.**
* **At the lowest level, testing verifies that the code works.**
* **At the next level, verification confirms that the application satisfies the requirements, and validation confirms that the application meets the customers’ needs.**
* **This process of working back up to the conceptual top of the application is called *integration* .**
* **Each of the tasks on the left corresponds to a task on the right with a similar level of abstraction.**
* **At the highest level, the initial concept corresponds to operation and maintenance.**
* **At the next level, the requirements correspond quite directly to verification and validation. Testing confirms that the design worked.**
* **SYSTEMS DEVELOPMENT LIFE CYCLE**
* **The software development life cycle ( SDLC ), is also called the application development life cycle.**
* **It covers all the tasks that in a software engineering project from start to finish: requirements, design, implementation,verification,deployment,maintenance.**
* **Waterfall model is one version of SDLC.**
* **The incremental waterfall model is basically just a series of SDLCs flattened out and possibly with some overlap. So one project starts before the previous one is completely finished.**
* **The two main ideas of SDLC**
* **The end of one project can feed directly into the next project in a never‐ending circle of life.**
* **The second new idea is that you can break down the basic steps in a lot more detail if you like.**
* **Tasks that are commonly represented as part of the SDLC as follows.**
* **Initiation —An initiator (often a customer, executive champion, or software manager) comes up with the initial idea.**
* **Concept development —The initiator, usually with help from others who might be interested in the project, explores the concept to see if it’s worthwhile and to evaluate possible alternatives.**
* **Preliminary planning —A project manager (PM) and technical lead are assigned to the project, and they start planning.**
* **If it’s a big project, the project might be broken into teams and team leads would be assigned.**
* **All these leaders make preliminary plans to estimate necessary resources such as staffing, computers, network, development tools etc.**
* **Requirements analysis —The team studies the user’s needs and creates requirement documents.**
* **Those may include text, pictures, use cases, prototypes, and long‐winded descriptions of business rules.**
* **It may also include UML diagrams showing application structure, user behavior, and anything else that helps the users understand what the team will be building.**
* **High‐level design —The team creates high‐level designs that specify major subsystems, data flow, database needs, and the rest of the application’s high‐level structure.**
* **Low‐level design —The team creates low‐level designs that explain how to build the application’s pieces.**
* **Development —The team writes the program code and follow good programming practices.**
* **They perform unit tests, regression tests, and system tests.**
* **They fix the bugs that appear and handle any change requests that are approved by the change committee.**
* **The team also prepares user documentation and training materials**
* **Acceptance testing —The customers finally get a chance to take the application for a test drive in its (almost) final form.**
* **After a few bug fixes and perhaps a small change or two, the customers agree that the application satisfies the requirements**
* **Deployment —The team rolls out the application.**
* **Maintenance —bug fixes and change requests.**
* **The team continues to track the application’s usefulness throughout its lifetime to determine whether it needs repair, enhancement, or replacement with a new version or with something completely different**
* **The maintenance team needs to figure out how to upgrade the application to the latest hardware and operating system, and how to dispose of the old hardware**
* **Review —The team uses metrics to assess the project and decide whether the development process can be improved in the future.**
* **Disposal —Eventually, the application’s usefulness comes to an end. During this stage, cleanup crew plans the application’s removal and possibly its replacement by something else.**
* **The crew needs to decide what data needs to be archived and how to protect it so that a hacker can’t break in and steal sensitive data.**
* **ITERATIVE MODEL**
* **ITERATIVE VERSUS PREDICTIVE**
* **Predictive models are ill‐suited to handle unexpected change.**
* **Predictive projects spend a lot of effort at the beginning , figuring out exactly what they will do. After you gather requirements and commit to a schedule, it’s hard to change course.**
* **Predictive models also don’t handle fuzzy requirements well.**
* **ITERATIVE MODEL**
* **Iterative models address build the application incrementally.**
* **They start by building the smallest program that is reasonably useful.**
* **Then they use a series of *increments* to add more features to the program until it’s finished.**
* **ITERATIVE MODEL**
* **ADVANTAGES OF ITERATIVE MODEL**
* **Each increment has a relatively small duration (compared to a predictive project), and so we’re committed to a smaller amount of work.**
* **Its possible to reorient the project before each new increment, so its less likely to need to cancel anything.**
* **Iterative models also handle fuzzy requirements reasonably well.**
* **ITERATIVE VERSUS INCREMENTAL**
* **An iterative project might not be incremental.**
* **Karl Scotland provides an interesting perspective on this issue.**
* **According to Karl Scotland ,the difference between iterative and incremental development is clear by considering the fidelity(completeness) of a project’s features.**
* **Consider we’re working on a project that provides three features. Here’s how we might use fidelity to describe different development approaches:**
* **Predictive -Provides all three features at the same time with full fidelity.**
* **Iterative -Initially provides all three features at a low (but usable) fidelity.**
* **Later iterations provide higher and higher fidelity until all the features are provided with full fidelity.**
* **Incremental -Initially provides the fewest possible features for a usable application, but all the features present are provided with full fidelity. Later versions add more features, always at full fidelity.**
* **Agile -Initially provides the fewest possible features at low fidelity. Later versions improve the fidelity of existing features and add new features. Eventually all the features are provided at full fidelity.**
* **ITERATIVE MODELS**
* **PROTOTYPES**
* **Two important facts about prototypes are**
* **They don’t need to work the same way the final application will.**
* **They don’t need to implement all the features of the final application**
* **After the customers experiment with the prototype, they can give the feedback to help refine the requirements.**
* **Types of Prototypes**
* **Throwaway prototypes- We use the prototype to study some aspect of the system and then you throw it away and write code from scratch**
* **Evolutionary prototypes-The prototype demonstrates some of the application’s features. As the project progresses, you refine those features and add new ones until the prototype morphs into the finished application.**
* **Incremental Prototyping- In *incremental prototyping* , we build a collection of prototypes of that separately demonstrate the finished application’s features and then combine the prototypes (or at least their code) to build the finished application**
* **PROTOTYPE -PROS**
* **Prototyping’s main benefits:**
* **Improved requirements —Prototypes allow customers to see what the finished application will look like. That lets them provide feedback to modify the requirements early in the project.**
* **PROTOTYPE -PROS**
* **Common vision —Prototypes let the customers and developers see the same preview of the finished application, so they are more likely to have a common vision of what the application should do and what it should look like.**
* **PROTOTYPE -PROS**
* **Better design —Prototypes (particularly vertical prototypes) let the developers quickly explore specific pieces of the application to learn what they involve. Prototypes also let developers test different approaches to see which one is best.**
* **PROTOTYPE -CONS**
* **Narrowing vision —People tend to focus on a prototype’s specific approach rather than on the problem it addresses.**
* **Customer impatience —A good prototype can make customers think that the finished application is just around the corner**
* **Schedule pressure —If customers see a prototype that they think is mostly done, they may not understand that you need another year to finish and may pressure you to shorten the schedule**
* **Raised expectations —Sometimes, a prototype may demonstrate features that won’t be included in the application.**
* **Attachment to code —Sometimes, developers become attached to the prototype’s code. That can make them follow the methods used by that code (or even reuse the code wholesale) even if a better design exists.**
* **Never‐ending prototypes —Throwaway prototypes are supposed to be built relatively quickly to provide fast feedback.**
* **Sometimes, developers spend far too much time refining a prototype to make it look better and include more features that aren’t actually necessary.**
* **The spiral model was first described in 1986 by Barry Boehm. He describes it as a “process model generator.” It uses a**
* **Risk‐driven approach to help project teams decide on what development approach to take for various parts of the project.**
* **If you don’t understand all the requirements, then you might use an iterative approach for developing them.**
* **In the first phase, you determine the objectives of the current cycle. You define any alternatives and constraints on the objectives.**
* **In the second phase ( risk analysis phase), risk analysis is performed to determine what the biggest risk factors are that could prevent you from achieving this cycle’s objectives.**
* **Resolve the risks and build a prototype to achieve your objectives.**
* **In the third phase ( engineering phase), use the prototype we just built to evaluate our solution.**
* **Some simulations and model specific problems to see if we’re on the right track.**
* **In the fourth phase ( evaluation phase), we evaluate the progress so far and make sure the project’s major stakeholders agree that the solution you came up with is correct and that the project should continue.**
* **If they decide you’ve made a mistake, you run another lap around the spiral to fix whatever problems remain. (You identify the missed objectives, evaluate alternatives, identify and resolve risks, and produce another prototype.)**
* **After you’re sure you’re on the right track, you plan the next trip around the spiral.**