Software Development Process

A software development process, also known as a software development life-cycle (SDLC), is a structure imposed on the development of a software product.

The activities involved in software development process are as follows:

* Communication and planning: This involves heavy communication and collaboration with the customers and other stakeholders and encompasses requirements gathering and other related activities. Customers typically have an abstract idea of what they want as an end result, but do not know what software should do. Skilled and experienced software engineers recognize incomplete, ambiguous, or even contradictory requirements at this point.
* Modeling: This activity encompasses the creation of models that allow the developers and the customers to better understand software requirements and the design that will achieve those requirements.
* Construction: This activity involves implementation, testing and documentation. Implementation is the part where the software programmers actually program the code for the project. Software testing ensures that errors in the code are recognized as soon as possible. Documentation helps in future maintenance and enhancement.
* Deployment and Maintenance: The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation. Maintenance involves discovery of faults and addition of new requirements.

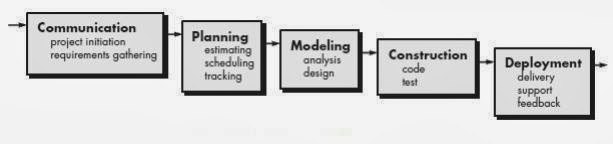
Software Development Process

Waterfall Model

The waterfall model is a systematic and sequential approach to software development. It is the oldest paradigm for software engineering.

The sequential phases in a waterfall model are as follows:

1. Communication
   * Project Initiation
   * Requirements Gathering
2. Planning
   * Estimation
   * Scheduling
   * Tracking
3. Modeling
   * Analysis
   * Design
4. Construction
   * Code
   * Test
5. Deployment
   * Delivery
   * Support
   * Feedback



The problems that are encountered when the waterfall model is applied are:

* After the project requirements are gathered in the first phase, there is no formal way to make changes to the project as requirements change or more information becomes available to the project team.
* Real projects rarely follow the sequential flow. Changes can cause confusion as the project team proceeds.
* It is often difficult for the customer to state all requirements explicitly.
* A working version of the program will not be available until late in the project time-span. So patience is required and a major blunder, if undetected until the working program is reviewed, can be disastrous.

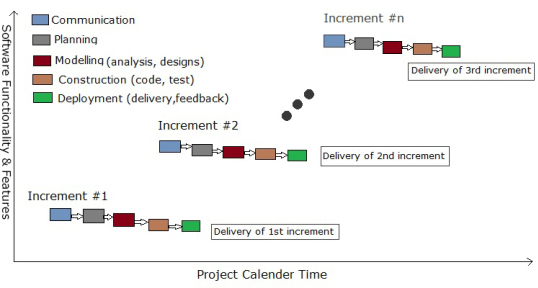
Incremental Model

The incremental build model is a method of software development where the model is designed, implemented and tested incrementally (a little more is added each time) until the product is finished. It involves both development and maintenance. The product is defined as finished when it satisfies all of its requirements. This model combines the elements of the waterfall model with the iterative philosophy of prototyping.

The product is decomposed into a number of components, each of which is designed and built separately (termed as builds). The first increment is often a core product in which basic requirements are addressed. The supplementary features (known or unknown) are delivered in later increments. Each component is delivered to the client when it is complete.

For each increment, the tasks involved are:

1. Communication
2. Planning
3. Modelling
4. Construction
5. Deployment



Advantages of using Incremental Model are as follows:

* Since, each component is delivered to the client when it is complete, partial utilization of the product is possible and a long development time can be avoided.
* It also avoids a large initial capital outlay and subsequent long waiting period.
* This model of development also helps ease the traumatic effect of introducing a completely new system all at once.
* Increments can be planned to manage technical risks.
* Initial product delivery is faster and costs lower.
* Clients can respond to features and review the product for any changes required.
* Testing and debugging is easier because relatively small changes are made in each increment.

The problems associated with Incremental Build Model are as follows:

* Resulting cost may exceed the cost of the organization.
* As additional functionality is added to the product, problems may arise related to system architecture which were not evident in earlier prototypes.

Evolutionary Development Model

Evolutionary Models are iterative models. Prototyping and Spiral Models are the two evolutionary models.

Prototyping

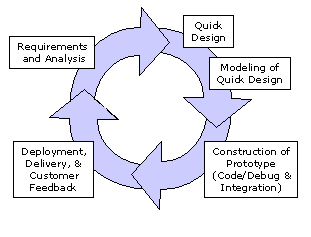
Prototyping paradigm may offer the best approach in following situations:

Often, a customer defines a set of general objectives for the software, but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction takes place. In such situations prototyping paradigm may offer the best approach.

The steps involved in prototyping are as follows:

1. Prototype starts with communication. The objectives and known requirements are identified.
2. A quick planning, modelling and design is done.
3. A working prototype is constructed.
4. The working prototype is deployed and evaluated.

The prototype can serve as the “first system”. The prototype can be thrown away, and the final product can be developed from scratch in a proper manner, or further refined prototypes can be built until it can work as a final product.



The advantages of prototyping model are as follows:

* Development can be done even without clear understanding of how the product should function.
* Time and cost for development is reduced.

The problems associated with prototyping are as follows:

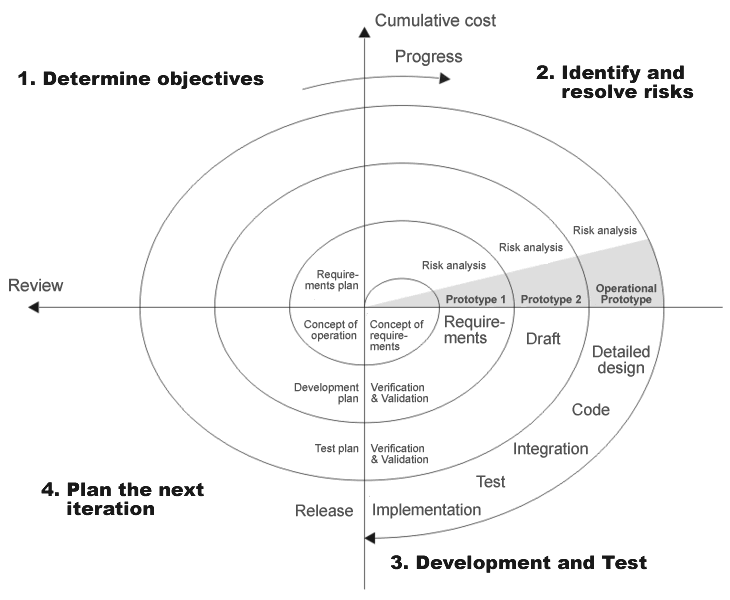
* Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions, preparation of incomplete specifications or the conversion of limited prototypes into poorly engineered final projects that are hard to maintain. Further, since a prototype is limited in functionality it may not scale well if the prototype is used as the basis of a final deliverable, which may not be noticed if developers are too focused on building a prototype as a model.
* User confusion of prototype and finished system: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished. (They are, for example, often unaware of the effort needed to add error-checking and security features which a prototype may not have.) This can lead them to expect the prototype to accurately model the performance of the final system when this is not the intent of the developers. Users can also become attached to features that were included in a prototype for consideration and then removed from the specification for a final system. If users are able to require all proposed features be included in the final system this can lead to conflict.
* Developer attachment to prototype: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture.
* Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex. When the prototype is thrown away the precisely developed requirements that it provides may not yield a sufficient increase in productivity to make up for the time spent developing the prototype. Users can become stuck in debates over details of the prototype, holding up the development team and delaying the final product.
* Expense of implementing prototyping: the start up costs for building a development team focused on prototyping may be high. Many companies have development methodologies in place, and changing them can mean retraining, retooling, or both. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.

Spiral Model

The key characteristic of a Spiral model is risk management at regular stages in the development cycle. In 1988, Barry Boehm published a formal software system development "spiral model," which combines some key aspect of the waterfall model and rapid prototyping methodologies, but provided emphasis in a key area many felt had been neglected by other methodologies: deliberate iterative risk analysis, particularly suited to large-scale complex systems.

The Spiral is visualized as a process passing through some number of iterations, with the four quadrant diagram representative of the following activities:

1. Formulate plans to: identify software targets, implement the program, clarify the project development restrictions
2. Risk analysis: an analytical assessment of selected programs, to consider how to identify and eliminate risk
3. Implementation of the project: the implementation of software development and verification



Risk-driven spiral model, emphasizing the conditions of options and constraints in order to support software reuse, software quality can help as a special goal of integration into the product development. However, the spiral model has some restrictive conditions, as follows:

1. The spiral model emphasizes risk analysis, and thus requires customers to accept this analysis and act on it. This requires both trust in the developer as well as the willingness to spend more to fix the issues, which is the reason why this model is often used for large-scale internal software development.
2. If the implementation of risk analysis will greatly affect the profits of the project, the spiral model should not be used.
3. Software developers have to actively look for possible risks, and analyze it accurately for the spiral model to work.

The first stage is to formulate a plan to achieve the objectives with these constraints, and then strive to find and remove all potential risks through careful analysis and, if necessary, by constructing a prototype. If some risks cannot be ruled out, the customer has to decide whether to terminate the project or to ignore the risks and continue anyway. Finally, the results are evaluated and the design of the next phase begins.

Basic Entities in Object Orientation

* Class: A class is a way to bind the data describing an entity and its associated functions together. It represents a group of similar objects broadly. It can be defined as a template that describes the behaviors/states that the objects of its type.
* Objects: An Object is an identifiable entity with some characteristics and behavior. An object is an instance of class.
* Attributes: Attributes (or data members) are the data-type properties that describe the characteristics of a class.
* Methods: A method (or member function) is a subroutine (or procedure or function) associated with an object, and which has access to its data, its member variables.
* Interface: An interface is a contract between a class and the outside world. When a class implements an interface, it promises to provide the behavior published by that interface. Methods form the object's interface with the outside world. The buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.
* Package: A package is a namespace for organizing classes and interfaces in a logical manner. Placing your code into packages makes large software projects easier to manage.
* Component:

Properties of Object Orientation

* Abstraction: Abstraction refers to the act of representing essential features without including background details or explanations. For instance, using a ‘switch board’, we need to know only its essential features that is what all switches are to be pressed and the background details like wiring and circuitry are hidden from us.
* Encapsulation: Encapsulation refers to wrapping up data and associated functions into one single unit(class).
* Data Hiding: Data hiding is a form of abstraction that allows only the relevant information to be exposed to the user.
* Polymorphism: Polymorphism is the ability for a message or data to be processed in more than one form.
* Inheritance: Inheritance is the capability of one class of things to inherit the capabilities or properties from another class.
* Message Passing: Message passing is a method by which an object sends data to another object or requests other object to invoke method.