**Software inspections**,

**Inspection** in software engineering, refers to peer review (industry best-practice for detecting software defects early and learning about software artifacts) of any work product by trained individuals who look for defects using a well-defined process.

**Inspection process**

- **Planning:** The inspection is planned by the moderator.
- **Overview meeting:** The author describes the background of the work product.
- **Preparation:** Each inspector examines the work product to identify possible defects.
- **Inspection meeting:** During this meeting the reader reads through the work product, part by part and the inspectors point out the defects for every part.
- **Rework:** The author makes changes to the work product according to the action plans from the inspection meeting.
- **Follow-up:** The changes by the author are checked to make sure everything is correct.

**Inspection roles**

During an inspection the following roles are used.

- **Author:** The person who created the work product being inspected.
- **Moderator:** This is the leader of the inspection. The moderator plans the inspection and coordinates it.
- **Reader:** The person reading through the documents, one item at a time. The other inspectors then point out defects.
- **Recorder/Scribe:** The person that documents the defects that are found during the inspection.
- **Inspector:** The person that examines the work product to identify possible defects.

**Clean room software development,**

- The cleanroom software engineering process is a software development process intended to produce software with a certifiable level of reliability.
- The focus of the cleanroom process is on defect prevention, rather than defect removal.
- The name "cleanroom" was chosen to invoke the cleanrooms used in the electronics industry to prevent the introduction of defects during the fabrication of semiconductors.

**Defect Testing,**

This test is intended to find areas where the program does not conform to its specification. Tests are designed to reveal the presence of defects in the system. When defects have been found in a program, these must be pinpointed and removed. This is called debugging. In fact, they are quite different. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

**Integrating Testing,**

Integration is a key software development life cycle (SDLC) strategy. Generally, small software systems are integrated and tested in a single phase, whereas larger systems involve several integration phases to build a complete system, such as integrating modules into low-level subsystems for integration with larger subsystems. Integration testing encompasses all aspects of a software system's performance, functionality and reliability.

Most unit-tested software systems are comprised of integrated components that are tested for error isolation due to grouping. Module details are presumed accurate, but prior to integration testing, each module is separately tested via partial component implementation, also known as a stub.

The three main integration testing strategies are as follows:

- **Big Bang**: all component are integrated together at **once**, and then tested. This is considered a high-risk approach because it requires proper documentation to prevent failure.
- **Bottom-Up**: Involves low-level component testing, followed by high-level components. Testing continues until all hierarchical components are tested. Bottom-up testing facilitates efficient error detection.
- **Top-Down**: Involves testing the top integrated modules first. Subsystems are tested individually. Top-down testing facilitates detection of lost module branch links.

**Object –Oriented testing,**

Object-Oriented Testing is a collection of testing techniques to verify and validate object-oriented software

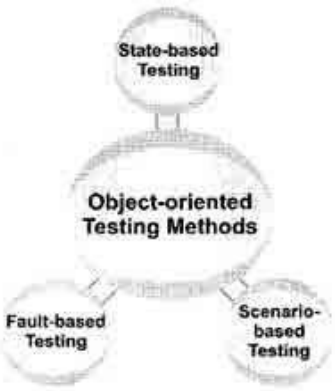
***Object Oriented Testing Activities***

- Review OOA and OOD models
- Class testing after code is written
- Integration testing within subsystems
- Validation testing based on OOA use- cases

***To complete the OOT cycle mention below testing are required.***

- Requirement Testing: Model review, prototype walkthrough, scenario testing
- Analysis and Design Testing: Model review, prototype walkthrough, scenario testing

- Code Testing: black-box, white-box, gray-box testing
- Integration Tests
- System Tests: function, installation, operation, stress, support testing
- User Testing: acceptance testing: alpha, beta, pilot testing



Object-oriented Testing Methods

**State-based testing** is used to verify whether the methods (a procedure that is executed by an object) of a class are interacting properly with each other. This testing seeks to exercise the transitions among the states of objects based upon the identified inputs.

**Fault-based testing** is used to determine or uncover a set of plausible faults. In other words, the focus of tester in this testing is to detect the presence of possible faults.

**Scenario-based testing** is used to detect errors that are caused due to incorrect specifications and improper interactions among various segments of the software. Incorrect interactions often lead to incorrect outputs that can cause malfunctioning of some segments of the software.

**Critical system validation.**

Critical systems are systems whose failure may lead to injury or loss of life, damage to the environment, unauthorized disclose of information or serious financial losses. There are three types of critical system:

i. **Safety-critical systems:** A system whose failure may result in injury, loss of life or serious environmental damage. E.g. a control system for a chemical manufacturing plant.
ii. **Mission-critical systems:** A system whose failure may result in the failure of some goal-directed activity. E.g. a mission-critical system is a navigational system for a spacecraft.
iii. **Business-critical systems:** A system whose failure may result in very high costs for the business using that system. E.g. a business-critical system is the customer accounting system in a bank. Business-critical systems may be affected by security-related failures.

**Validation perspectives**

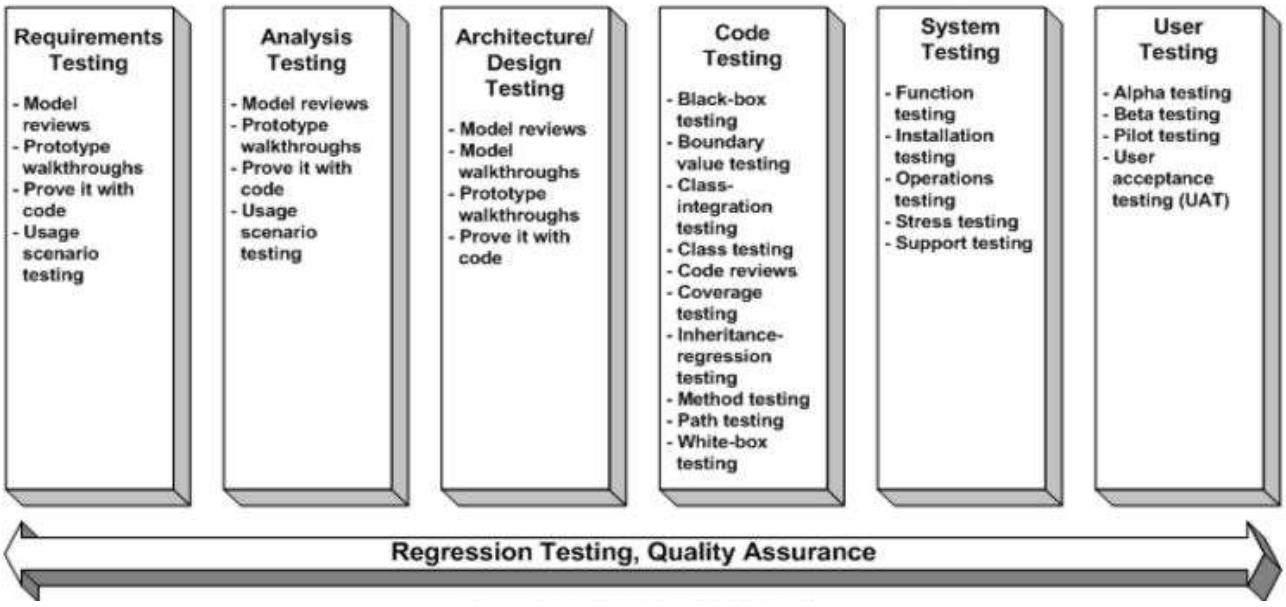
i. **Reliability validation**: Does the measured reliability of the system meet its specification? Is the reliability of the system good enough to satisfy users?
ii. **Safety validation**: Does the system always operate in such a way that accidents do not occur or that accident consequences are minimised?
iii. **Security validation**: Is the system and its data secure against external attack?

**Validation techniques**

i. **Static techniques**: Design reviews and program inspections, Mathematical arguments and proof
ii. **Dynamic techniques**: Statistical testing, Scenario-based testing, Run-time checking
iii. **Process validation**: Design development processes that minimise the chances of process errors that might compromise the dependability of the system

**Object Oriented Testing methods:**

Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.




**Unit Testing:**

- In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free.
- Unit testing is the responsibility of the application engineer who implements the structure.

**Subsystem Testing:**

- This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction of the subsystem with the outside.
- Subsystem tests can be used as regression tests for each newly released version of the subsystem.

**System Testing:**

- System testing involves testing the system as a whole and is the responsibility of the quality-assurance team. The team often uses system tests as regression tests when assembling new releases.

**Object-Oriented Testing Techniques:**

**Grey Box Testing:**

The different types of test cases that can be designed for testing object-oriented programs are called grey box test cases. Some of the important types of grey box testing are:

- **State model based testing:** This encompasses state coverage, state transition coverage, and state transition path coverage.
- **Use case based testing:** Each scenario in each use case is tested.
- **Class diagram based testing:** Each class, derived class, associations, and aggregations are tested.
- **Sequence diagram based testing:** The methods in the messages in the sequence diagrams are tested.

**Techniques for Subsystem Testing:**

The two main approaches of subsystem testing are:

- **Thread based testing:** All classes that are needed to realize a single use case in a subsystem are integrated and tested.
- **Use based testing:** The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

**Categories of System Testing:**

- **Alpha testing:** This is carried out by the testing team within the organization that develops software.
- **Beta testing:** This is carried out by select group of co-operating customers.
- **Acceptance testing:** This is carried out by the customer before accepting the deliverables.

**Black-box testing:** Testing that verifies the item being tested when given the appropriate input provides the expected results.

**Boundary-value testing:** Testing of unusual or extreme situations that an item should be able to handle.

**Class testing:** The act of ensuring that a class and its instances (objects) perform as defined.

**Component testing:** The act of validating that a component works as defined.

**Inheritance-regression testing:** The act of running the test cases of the super classes, both direct and indirect, on a given subclass.

**Integration testing:** Testing to verify several portions of software work together.

**Model review:** An inspection, ranging anywhere from a formal technical review to an informal walkthrough, by others who were not directly involved with the development of the model.

**Path testing:** The act of ensuring that all logic paths within your code are exercised at least once.

**Regression testing:** The acts of ensuring that previously tested behaviours still work as expected after changes have been made to an application.

**Stress testing:** The act of ensuring that the system performs as expected under high volumes of transactions, users, load, and so on.

**White-box testing:** Testing to verify that specific lines of code work as defined. Also referred to as clear-box testing.

**Technical review:**

A quality assurance technique in which the design of your application is examined critically by a group of your peers. A review typically focuses on accuracy, quality, usability, and completeness. This process is often referred to as a walkthrough, an inspection, or a peer review.

**User interface testing:**

The testing of the user interface (UI) to ensure that it follows accepted UI standards and meets the requirements defined for it. Often referred to as graphical user interface (GUI) testing.**Software Quality Attributes**

Correctness, Reliability, Adequacy, Learnability, Robustness, Maintainability, Readability, Extensibility, Testability, Efficiency, Portability.

**Correctness:** The correctness of a software system refers to:
- Agreement of program code with specifications
- Independence of the actual application of the software system.

The **correctness** of a program becomes especially critical when it is embedded in a complex software system.
**Reliability:** Reliability of a software system derives from Correctness and Availability.
The behavior over time for the fulfillment of a given specification depends on the reliability of the software system.

**Reliability** of a software system is defined as the probability that this system fulfills a function (determined by the specifications) for a specified number of input trials under specified input conditions in a specified time interval (assuming that hardware and input are free of errors).

A software system can be seen as reliable if this test produces a low error rate (i.e., the probability that an error will occur in a specified time interval.)

The error rate depends on the frequency of inputs and on the probability that an individual input will lead to an error.

**Adequacy: Factors for the requirement of Adequacy:**

**-** The input required of the user should be limited to only what is necessary. The software system should expect information only if it is necessary for the functions that the user wishes to carry out. The software system should enable flexible data input on the part of the user and should carry out plausibility checks on the input. In dialog-driven software systems, we vest particular importance in the uniformity, clarity and simplicity of the dialogs.

**-** The performance offered by the software system should be adapted to the wishes of the user with the consideration given to extensibility; i.e., the functions should be limited to these in the specification.

**- The results produced by the software system:** The results that a software system delivers should be output in a clear and wellstructured form and be easy to interpret. The software system should afford the user flexibility with respect to the scope, the degree of detail, and the form of presentation of the results. Error messages must be provided in a form that is comprehensible for the user.

**Learnability:** Learnability of a software system depends on:

- The design of user interfaces- The clarity and the simplicity of the user instructions (tutorial or user manual).

The user interface should present information as close to reality as possible and permit efficient utilization of the software's failures.

The user manual should be structured clearly and simply and be free of all dead weight. It should explain to the user what the software system should do, how the individual functions are activated, what relationships exist between functions, and which exceptions might arise and how they can be corrected. In addition, the user manual should serve as a reference that supports the user in quickly and comfortably finding the correct answers to questions.

**Robustness:** Robustness reduces the impact of operational mistakes, erroneous input data, and hardware errors.

**<u>Defect can be categorized into the following:</u>**
**Wrong:** When requirements are implemented not in the right way. This defect is a variance from the given specification. It is Wrong!
**Missing:** A requirement of the customer that was not fulfilled. This is a variance from the specifications, an indication that a specification was not implemented, or a requirement of the customer was not noted correctly.
**Extra:** A requirement incorporated into the product that was not given by the end customer. This is always a variance from the specification, but may be an attribute desired by the user of the product. However, it is considered a defect because it's a variance from the existing requirements.
**ERROR:** An error is a mistake, misconception, or misunderstanding on the part of a software developer. In the category of developer we include software engineers, programmers, analysts, and testers. For example, a developer may misunderstand a de-sign notation, or a programmer might type a variable name incorrectly – leads to an Error. It is the one which is generated because of wrong login, loop or due to syntax. Error normally arises in software; it leads to change the functionality of the program.
**BUG:** A bug is the result of a coding error. An Error found in the development environment before the product is shipped to the customer. A programming error that causes a program to work poorly, produce incorrect results or crash. An error in software or hardware that causes a program to malfunction. Bug is terminology of Tester.

**FAILURE:** A failure is the inability of a software system or component to perform its required functions within specified performance requirements. When a defect reaches the end customer it is called a Failure. During development Failures are usually observed by testers.

**FAULT:** An incorrect step, process or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner. A fault is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification. It is the result of the error.

The software industry can still not agree on the definitions for all the above. In essence, if you use the term to mean one specific thing, it may not be understood to be that thing by your audience.

**Component based software**

- An individual software component is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data).
- Sometimes called componentware, software designed to work as a component of a larger application. A good analogy is the way personal computers are built up from a collection of standard components: memory chips, CPUs, buses, keyboards, mice, disk drives, monitors, etc.

**Distributed software**

- Distributed applications (distributed apps) are applications or software that runs on multiple computers within a network at the same time and can be stored on servers or with cloud computing.
- Unlike traditional applications that run on a single system, distributed applications run on multiple systems simultaneously for a single task or job.

**Embedded software**

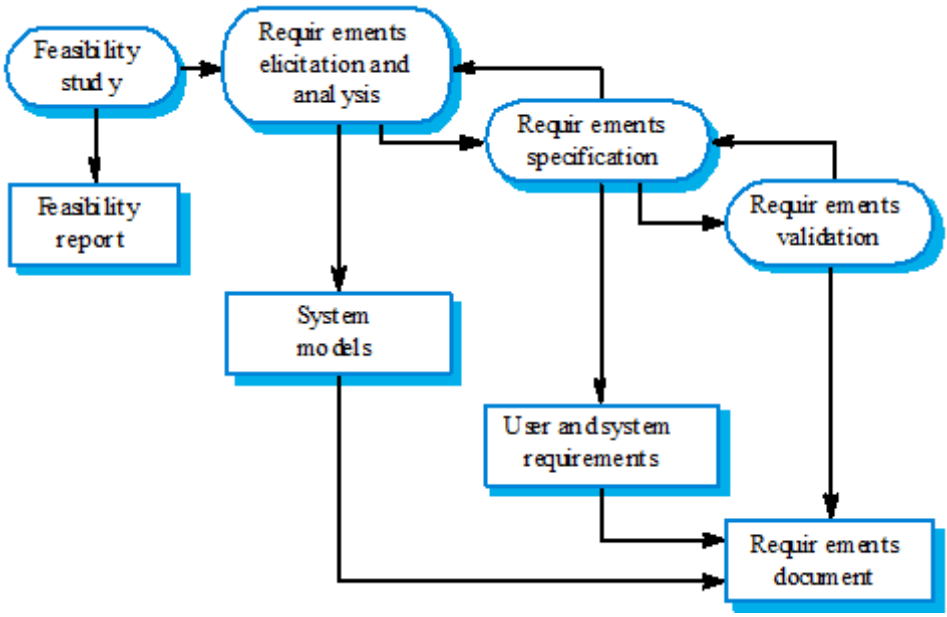
- Embedded software is a piece of software that is embedded in hardware or non-PC devices.
- It is written specifically for the particular hardware that it runs on and usually has processing and memory constraints because of the device's limited computing capabilities.
- Examples: dedicated GPS devices, factory robots, some calculators and even modern smartwatches.

**Realtime software design**

- A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced.
  i. A **soft real-time system** is a system whose operation is degraded if results are not produced according to the specified timing requirements.
  ii. A **hard real-time system** is a system whose operation is incorrect if results are not produced according to the timing specification.
- Designing embedded software systems whose behaviour is subject to timing constraints
- System elements: sensor control processes, Data processor, and Actuator control
- Real-time system correctness depends not just on what the system does but also on how fast it reacts
- A general RT system model involves associating processes with sensors and actuators
- Real-time systems architectures are usually designed as a number of concurrent processes
- Real-time executives are responsible for process and resource management.
- Monitoring and control systems poll sensors and send control signal to actuators
- Data acquisition systems are usually organised according to a producer consumer model

**R-T systems design process**

- Identify the stimuli to be processed and the required responses to these stimuli
- For each stimulus and response, identify the timing constraints
- Aggregate the stimulus and response processing into concurrent processes. A process may be associated with each class of stimulus and response
- Design algorithms to process each class of stimulus and response. These must meet the given timing requirements
- Design a scheduling system which will ensure that processes are started in time to meet their deadlines
- Integrate using a real-time executive or operating system




**Requirement Engineering Process**

**Feasibility study**: organizational objectives, technology, budget

**Elicitation and analysis:** requirements elicitation or requirements discovery. Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints. May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*

**Requirements validation**: Concerned with demonstrating that the requirements define the system that the customer really wants.

- ✓ **Requirements checking**: Validity, Consistency, Completeness, Realism, Verifiability
- ✓ **Requirements validation techniques**: Requirements reviews, Prototyping, Test-case generation

**Agile vs Prototyping**

- Prototype is not a complete system itself. Many details are also not built in the prototype model. Basic goal of Prototype model is to provide a system which gives overall functionality. Agile is used to plan quickly, develop quickly, release quickly and revise quickly.
- Agile Software Development to build real software products that I ship to paying customers. prototypes in order to learn something with no intention of shipping the prototype as a product
- The Agile is a way of working – a culture while prototyping is a technique to demonstrate a concept or an idea.
- Doing a prototype can be part of the agile software development, but not vice versa!

**Re-engineering and Reverse engineering**

- **Reverse engineering** is finding out how a product works from the finished product. **Re-engineering** is to examine the finished product and build it again, but better.
- **Reverse Engineering is** trying to recreate the source code from the compiled code. That is trying to figure out how a piece of software works given only the final system. **Re-Engineering** on the other hand is creating a new piece of software with similar functionality as an existing one. But you may be "improving" the way it was built.
- **RE-ENGINEERING** may be useful for the modification of the legacy code or software. **REVERSE-ENGINEERING** : Any activity that requires program understanding at any level may fall within the scope of reverse engineering.
- **Reverse engineering** is to take a bridge apart to see how it was built. **Re-engineering** is to throw a bridge away and rebuild it from scratch.
- **Re-engineering** means designing something again, perhaps from scratch. **Reverse engineering** means trying and understanding the inner workings of an artifact, most often without the help of explanatory material (such as documentation, drawings etc.).

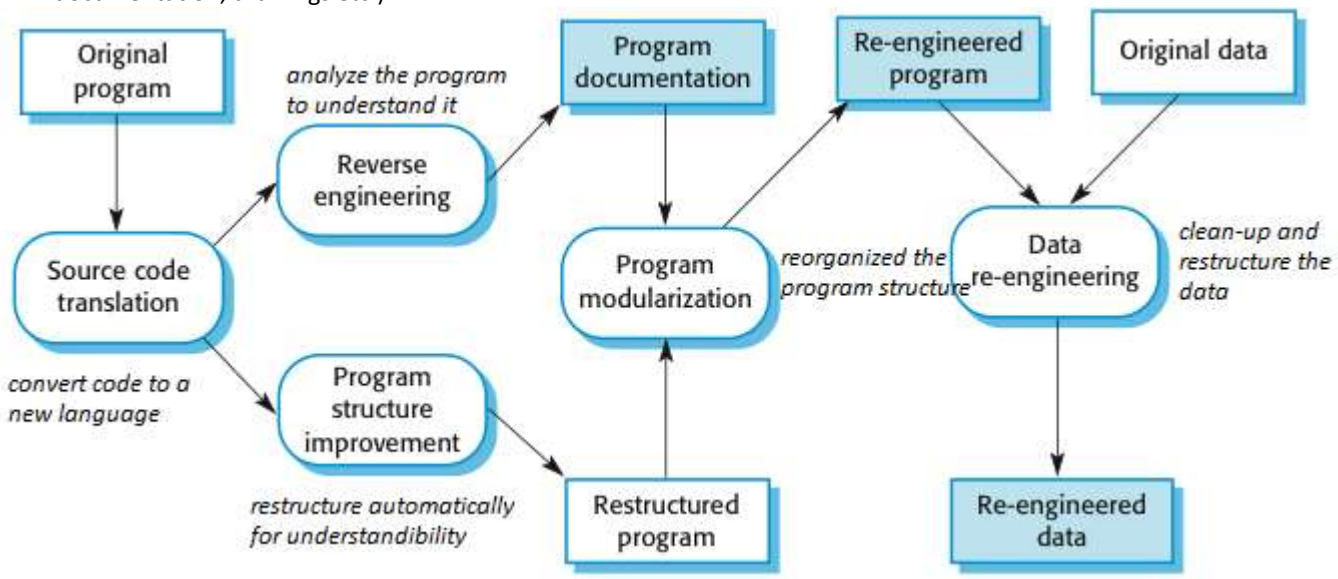



**Fig. Re-Engineering Process**

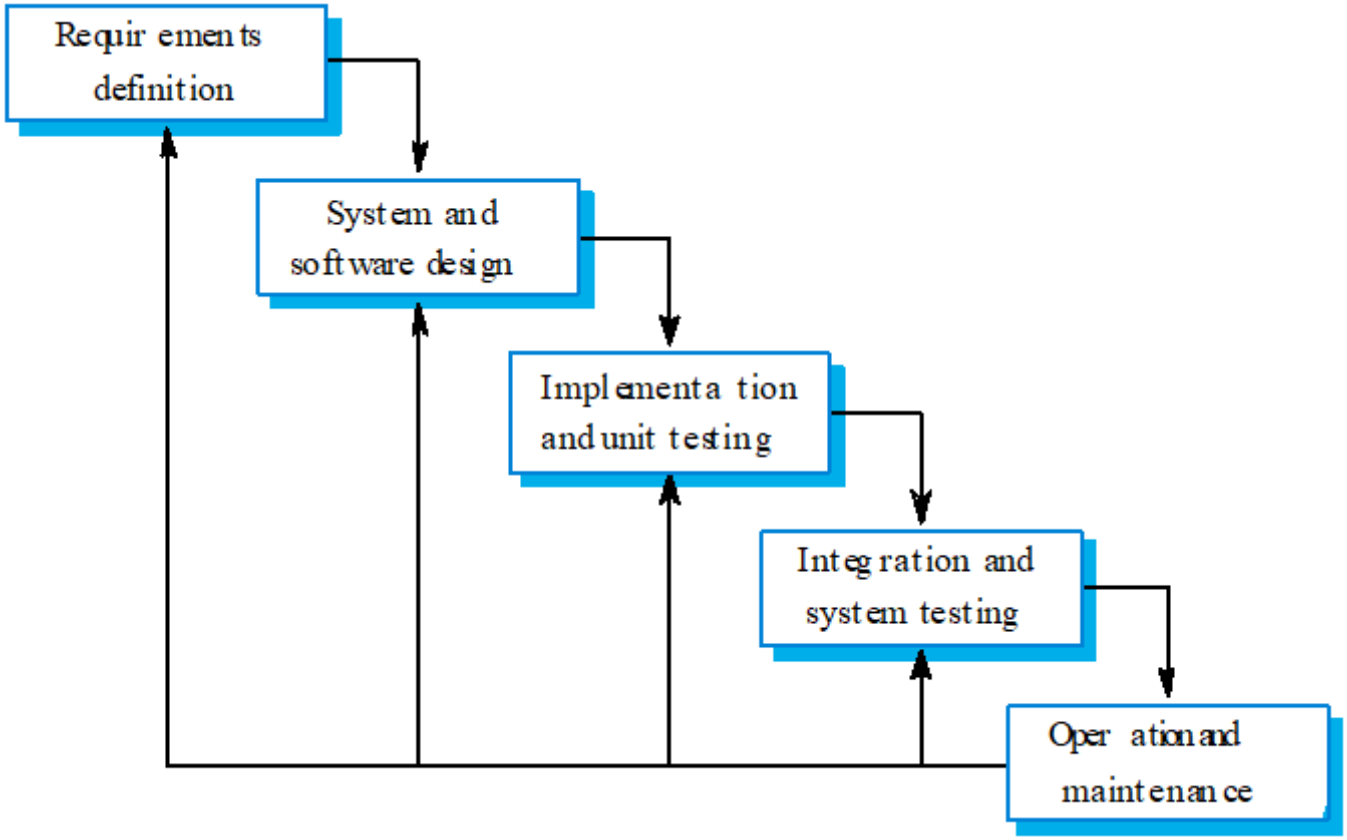



Fig. Waterfall Model

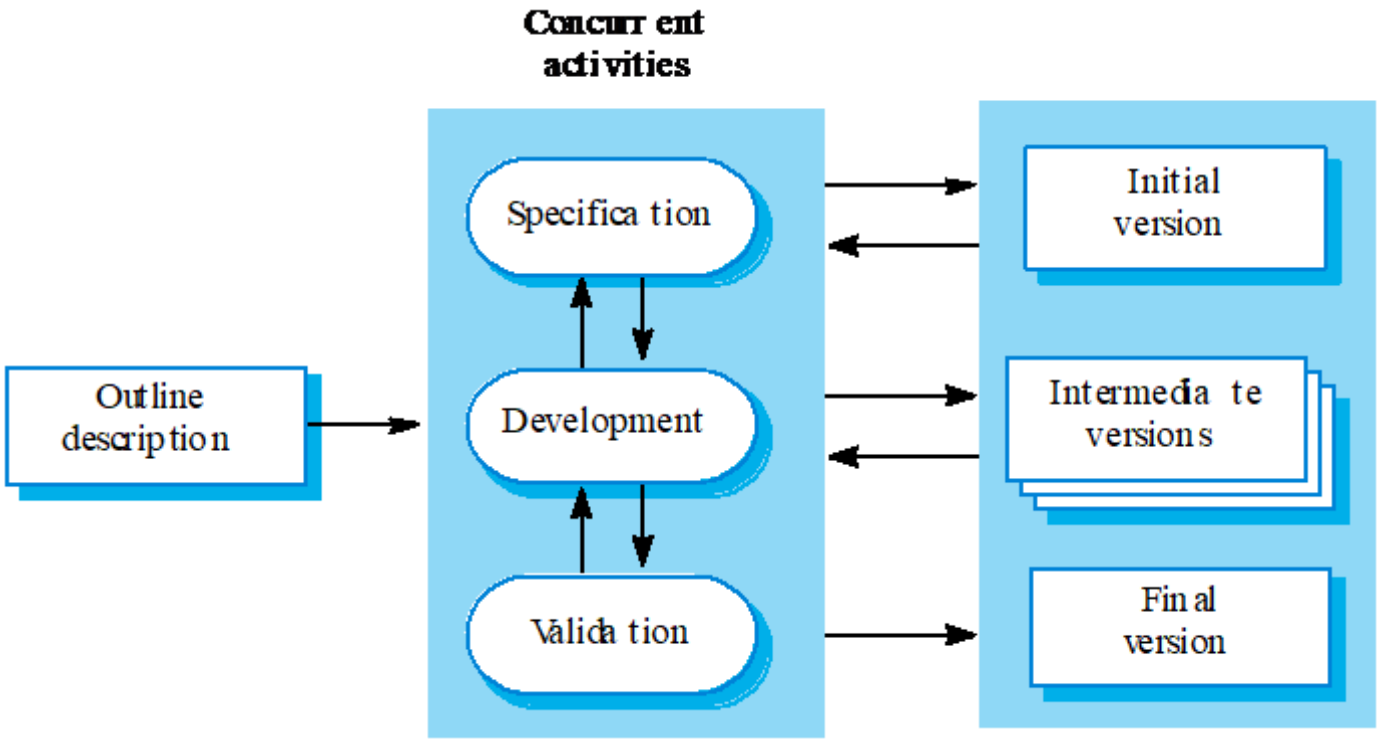



Fig. Evolutionary Development

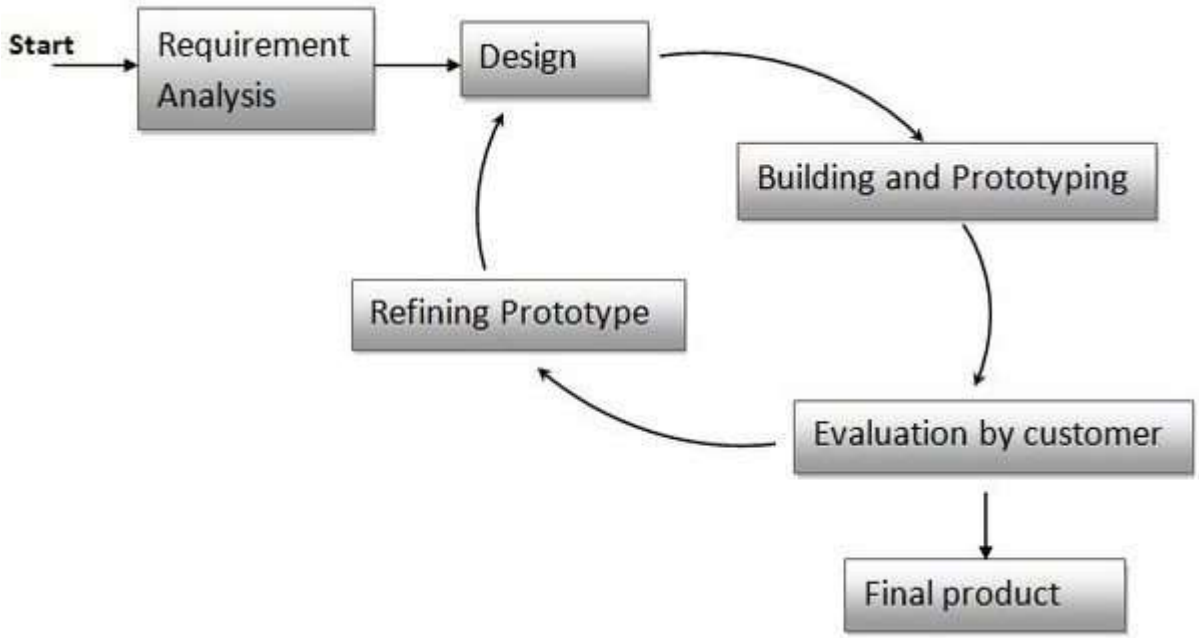



**Fig. Prototyping Model**

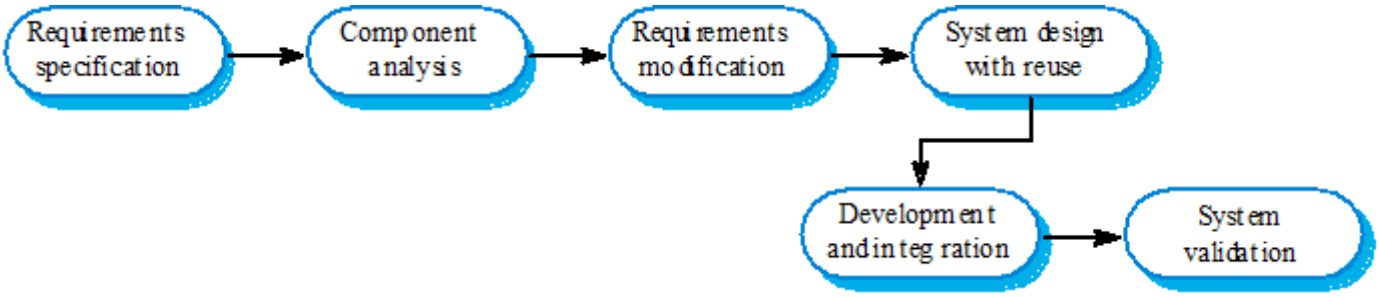



**Fig. Reuse-oriented development**

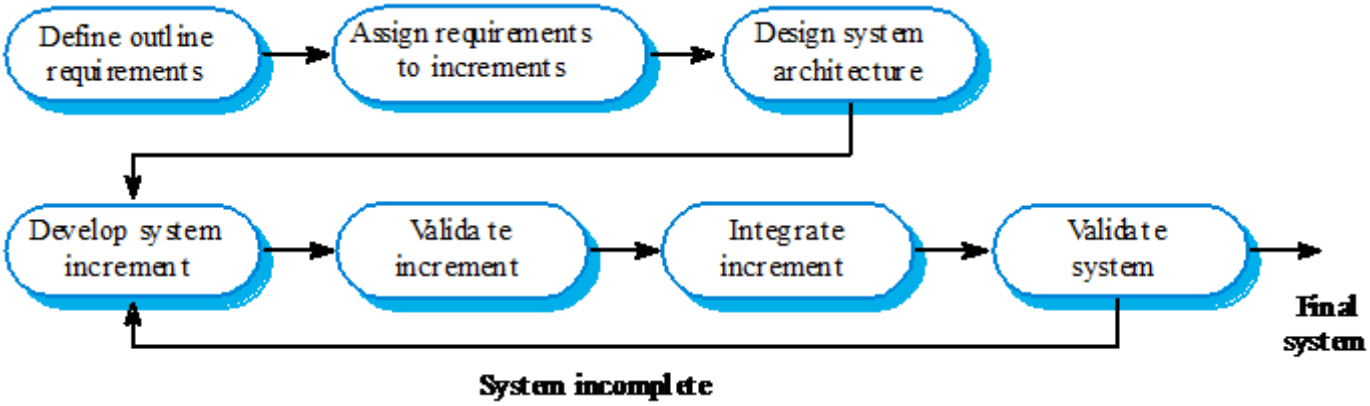



**Fig. Incremental development**

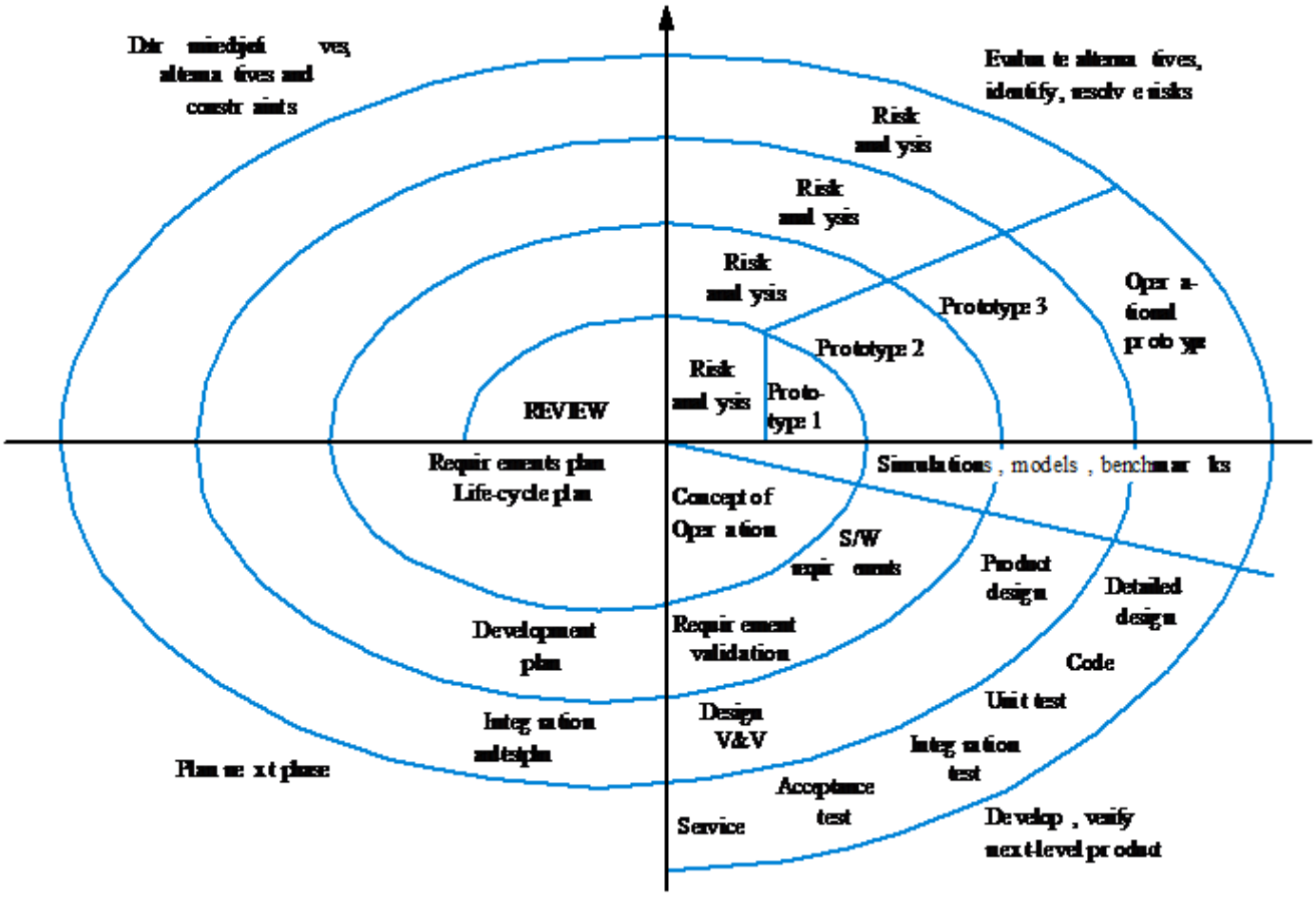



**Fig. Spiral model of the software process**

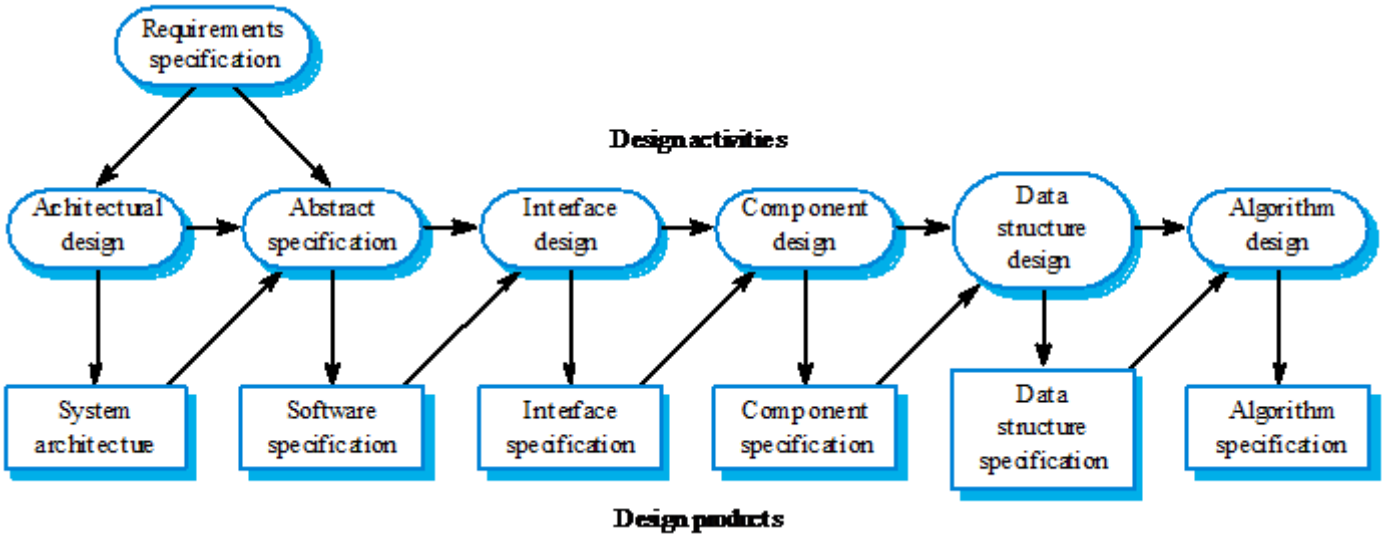



**Fig. The software design process**


**Fig. The debugging process**

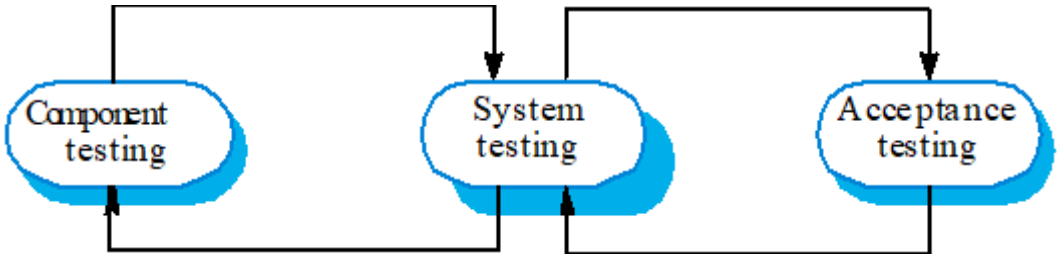



**Fig. The testing process**

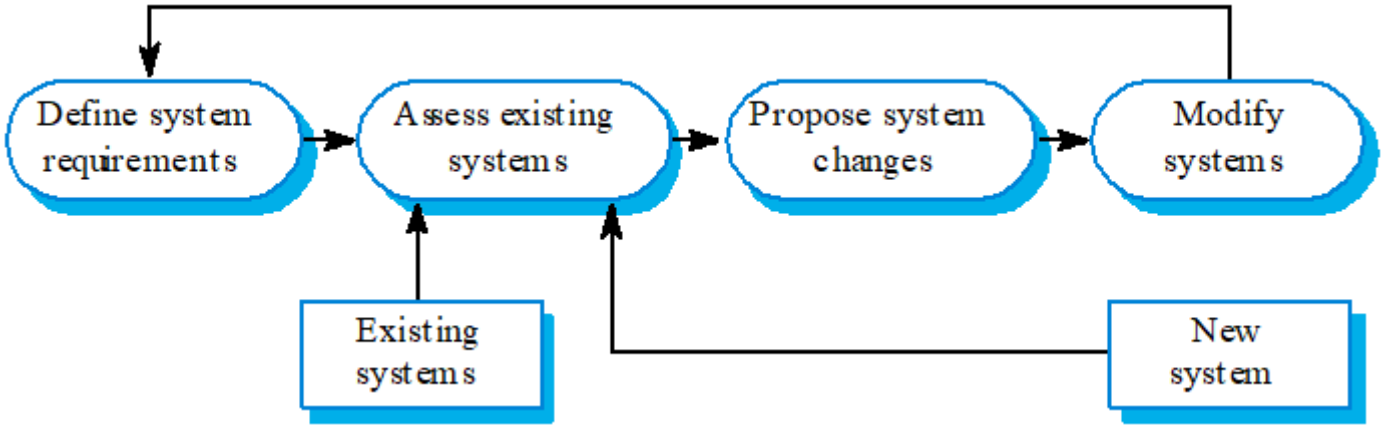
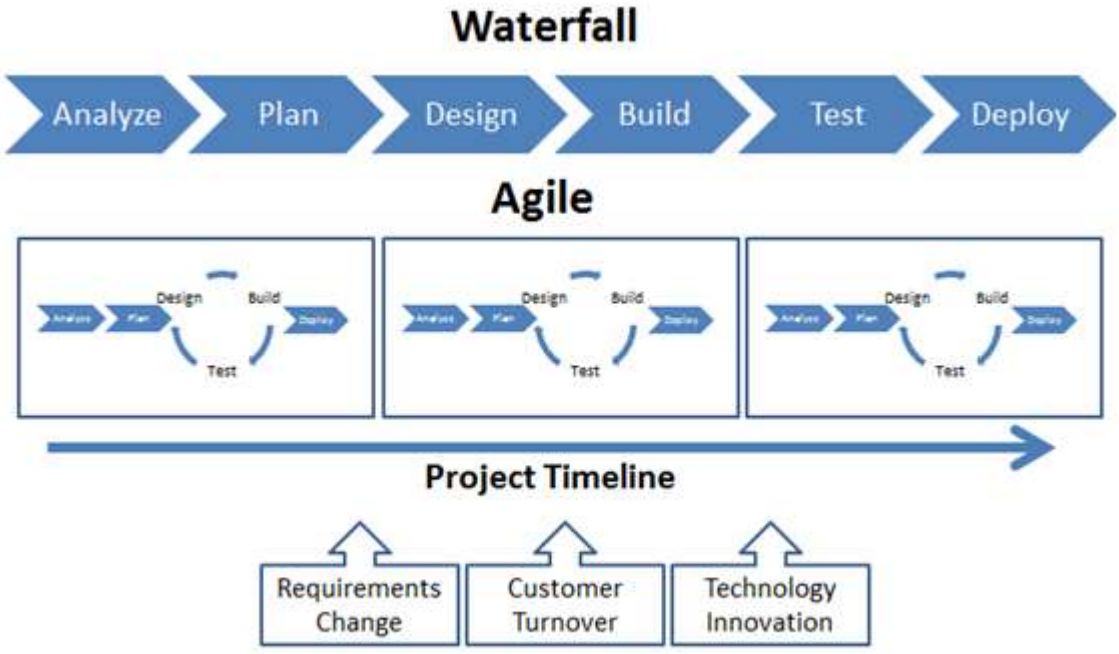



**Fig. System evolution**



## Comparison between Software Models

| S.N. | Parameter / Process Model | Waterfall Model | Incremental Model | Prototype Model | RAD Model | Spiral Model | Agile Model | XP programming |
|---|---|---|---|---|---|---|---|---|
| 1 | **Clear Requirement Specifications** | Initially | Initially | At medium level | Initially | Initially | Change incrementally | Initially |
| 2 | **Feedback from user** | Not required | Not required | Required | Not required | Not required | Not required | Required |
| 3 | **Speed to change** | Less speed of changing | Highly | Moderately | Not defined | Highly | Highly | Highly |
| 4 | **Predictability** | Less prediction | Less prediction | Highly Predicting | Less prediction | Medium | Highly Predicting | Highly Predicting |
| 5 | **Risk identification** | At first risk is identified | Not identified | Not identified | Not identified | Identified | Identified | Identified |
| 6 | **Practically implementation** | Not implemented | Less implemented | Moderately implemented | Not implemented | Moderately implemented | Mostly used | Mostly used |
| 7 | **Operation** | Systematic sequence | Iterative sequence | Priority on customer feedback | Use readymade component | Identification of risk at each stage | Highly customer satisfaction and incremental development | Customer satisfaction and incremental development |
| 8 | **Understandability** | Simple | Intermediate | Intermediate | Intermediate | Hard | Much complex | Intermediate |
| 9 | **Precondition** | Requirement clearly defined | Core product should clearly define | Clear idea of Quick Design | Clean idea of Reuse component | No | No | No |
| 10 | **Usability** | Basic | Medium | High | Medium | Medium | Most use now a day | Medium |
| 11 | **Customer priority** | Not required after starting | Not required after starting | Required before and after starting | Not required after starting | Required before and after starting | Required before and after starting | Required before and after starting |
| 12 | **Industry approach** | Basic | Basic | Medium | Medium | Medium | High | Medium |
| 13 | **Development and Production Cost** | Low | Low | High | Very High | Expensive | Very Expensive | High |
| 14 | **Resource organization** | Yes | Yes | Yes | Yes | No | No | Yes |
| 15 | **Elasticity** | No | No | Yes | Yes | No | Very high | Medium |