RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Computer Science and Engineering, VII-Semester

CS701 Software Architectures

Pre-Requisite: Software Engineering

Course Outcomes:
After completing the course student should be able to:

1. Describe the Fundamentals of software architecture, qualities and terminologies.
2. Understand the fundamental principles and guidelines for software architecture design, architectural styles, patterns, and frameworks.
3. Use implementation techniques of Software architecture for effective software development.
4. Apply core values and principles of software architectures for enterprise application development.

Course Contents:

Unit 1. Overview of Software development methodology and software quality model, different models of software development and their issues. Introduction to software architecture, evolution of software architecture, software components and connectors, common software architecture frameworks, Architecture business cycle – architectural patterns – reference model.

Unit 2. Software architecture models: structural models, framework models, dynamic models, process models. Architectures styles: dataflow architecture, pipes and filters architecture, call-and return architecture, data-centered architecture, layered architecture, agent based architecture, Micro-services architecture, Reactive Architecture, Representational state transfer architecture etc.

Unit 3. Software architecture implementation technologies: Software Architecture Description Languages (ADLs), Struts, Hibernate, Node JS, Angular JS, J2EE – JSP, Servlets, EJBs; middleware: JDBC, JNDI, JMS, RMI and CORBA etc. Role of UML in software architecture.

Unit 4. Software Architecture analysis and design: requirements for architecture and the life-cycle view of architecture design and analysis methods, architecture-based economic analysis: Cost Benefit Analysis Method (CBAM), Architecture Tradeoff Analysis Method (ATAM). Active Reviews for Intermediate Design (ARID), Attribute Driven Design method (ADD), architecture reuse, Domain – specific Software architecture.

Unit 5. Software Architecture documentation: principles of sound documentation, refinement, context diagrams, variability, software interfaces. Documenting the behavior of software elements and software systems, documentation package using a seven-part template.

UNIT 1

- Overview of Software development methodology and software quality model,
- different models of software development and their issues.
- Introduction to software architecture,
- evolution of software architecture,
- software components and connectors,
- common software architecture frameworks,
- Architecture business cycle – architectural patterns – reference model.

In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called 'architecture.'—

Martin Fowler

Software Architecture

♦ A software architect makes important decisions regarding the software that goes on to define its overall integrity. A good software architecture helps define attributes such as performance, quality, scalability, maintainability, manageability, and usability.

♦ Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

1. It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
2. Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of −
3. Selection of structural elements and their interfaces by which the system is composed.
4. Behavior as specified in collaborations among those elements.
5. Composition of these structural and behavioral elements into large subsystem.
6. Architectural decisions align with business objectives.
7. Architectural styles guide the organization.

♦ The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.

♦ We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In **Architecture**, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

Why Does Software Architecture Matter?

- ♦ An organized software architecture helps to ensure the longevity of the software's internal quality.

- ♦ Consider two similar products. Both are launched within a month-long gap and aims to add new features when they complete three months.

- ♦ There are two scenarios:

1. Product A launched in Jan 2021. This project supports a messy source code because the development team wanted to launch and monopolize the market as early as possible.
2. Product B launched in March 2021. This project has a software architecture that is well-structured and organized. The development team works on the design and architectural decisions early in the process and prioritizes quality over faster launch.
3. **Which Product will be more successful: A or B?**
4. Product A might monopolize the market initially and convert better. However, product adoption will eventually subside because the messy code will lead to technical debt pileups. These pileups will, in turn, make it challenging to introduce new updates and bug fixes on the fly.
5. Product B might have a market entry gap, but it will be easier to maintain a faster shipping cadence. The customer needs will be looked after without breaking the shipping cadence, thus making for a larger win.

Goals of Architecture

- ♦ The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.

- ♦ Some of the other goals are as follows −

1. Expose the structure of the system, but hide its implementation details.
2. Realize all the use-cases and scenarios.
3. Try to address the requirements of various stakeholders.
4. Handle both functional and quality requirements.
5. Reduce the goal of ownership and improve the organization's market position.
6. Improve quality and functionality offered by the system.
7. Improve external confidence in either the organization or system.

Limitations

- ♦ Software architecture is still an emerging discipline within software engineering. It has the following limitations −

1. Lack of tools and standardized ways to represent architecture.
2. Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
3. Lack of awareness of the importance of architectural design to software development.
4. Lack of understanding of the role of software architect and poor communication among stakeholders.
5. Lack of understanding of the design process, design experience and evaluation of design.

Role of Software Architect

- ♦ A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas −

- ♦ **Design Expertise**

1. Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.
2. Lead the development team and coordinate the development efforts for the integrity of the design.
3. Should be able to review design proposals and trade off among themselves.

- ♦ **Domain Expertise**

1. Expert on the system being developed and plan for software evolution.
2. Assist in the requirement investigation process, assuring completeness and consistency.
3. Coordinate the definition of domain model for the system being developed.

- ♦ **Technology Expertise**

1. Expert on available technologies that helps in the implementation of the system.
2. Coordinate the selection of programming language, framework, platforms, databases, etc.

- ♦ **Hidden Role of Software Architect**

1. Facilitates the technical work among team members and reinforcing the trust relationship in the team.
2. Information specialist who shares knowledge and has vast experience.
3. Protect the team members from external forces that would distract them and bring less value to the project.

Good Software Architecture Characteristics

The noteworthy characteristics of software architecture include:

1. **Uninterrupted Functionality** The software system performs as intended without any bugs or interruptions.
2. **Reliability** The software system performs optimally, irrespective of the environment and the number of inputs given.

3. Maintainability It is efficient and straightforward to introduce innovative changes to the software without interrupting the existing system's functionality.

4. Security The software is secure from all types of attacks, whether external or internal.

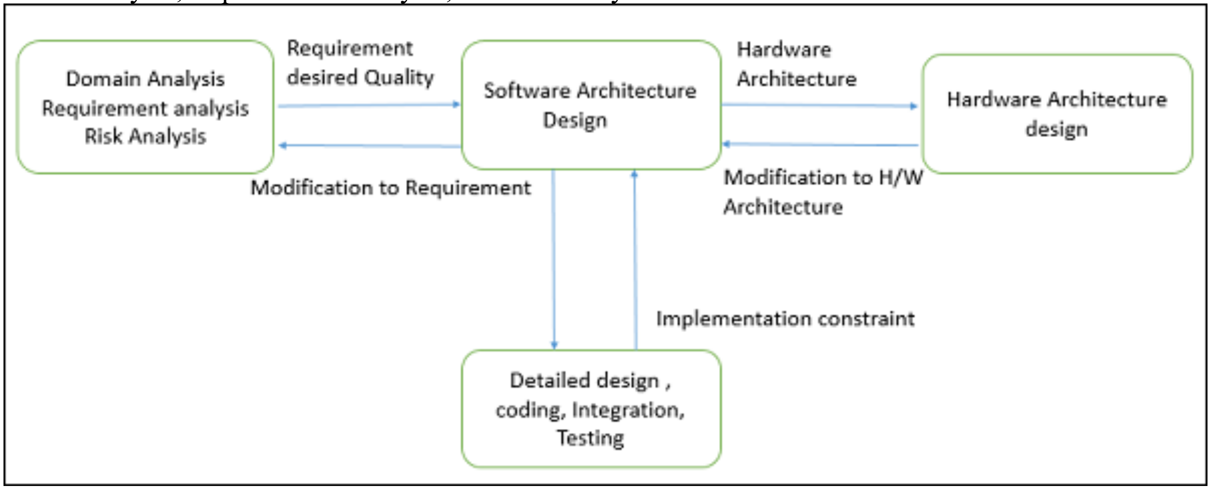
5. Minimal Technical Debt The source code is clean and organized rather than messy or full of anti-patterns. This reduces the code refactoring efforts and corresponding technical debt.

6. Testability Ensures quality software testing as it enables faster bugs identification. This, in turn, helps ensure faster time to market of the workable software.

7. Modularity With a good software architecture in place, it is easier to divide the software system into smaller, more manageable modules. This also helps when introducing microservices architecture when scaling up operations.

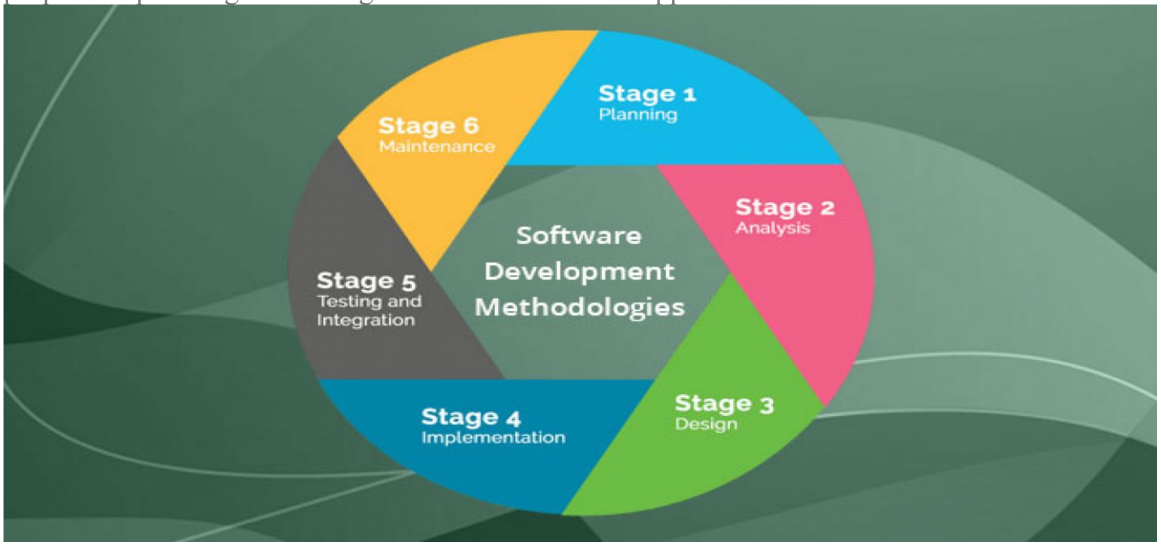
Software Design

Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows –

- To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.
- It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.
- 
Domain Analysis
Requirement analysis
Risk Analysis
Requirement
desired Quality
Modification to Requirement
Software Architecture Design
Hardware Architecture
Modification to H/W Architecture
Hardware Architecture design
Implementation constraint
Detailed design , coding, Integration, Testing

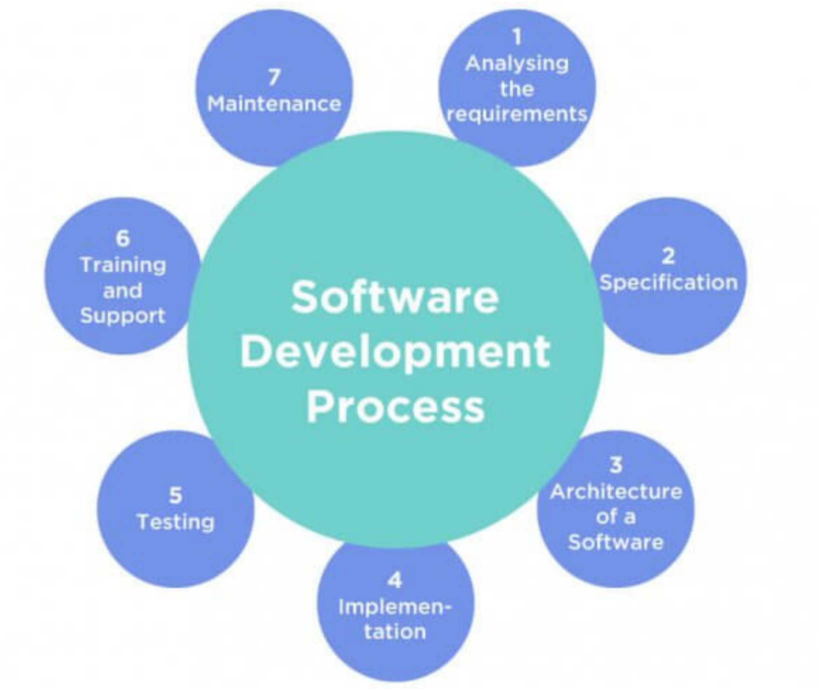
Overview of Software development methodology

What is a software development methodology?
Software development methodology refers to **structured processes involved when working on a project**. The goal is to provide a systematic approach to software development. Software Development Methodologies also called as the System Development Methodologies or in short a Software Process is a set of software development activities that are divided into phases for the purpose of planning and management of software and application.



Systems Development Life Cycle (SDLC)

The systems development life cycle is a conservative system that heavily bases on the stages of the product development process. It's similar to Waterfall – you can't jump from one step to another or return to the completed stage.


1 Analysing the requirements
2 Specification
3 Architecture of a Software
4 Implemen-tation
5 Testing
6 Training and Support
7 Maintenance
Software Development Process

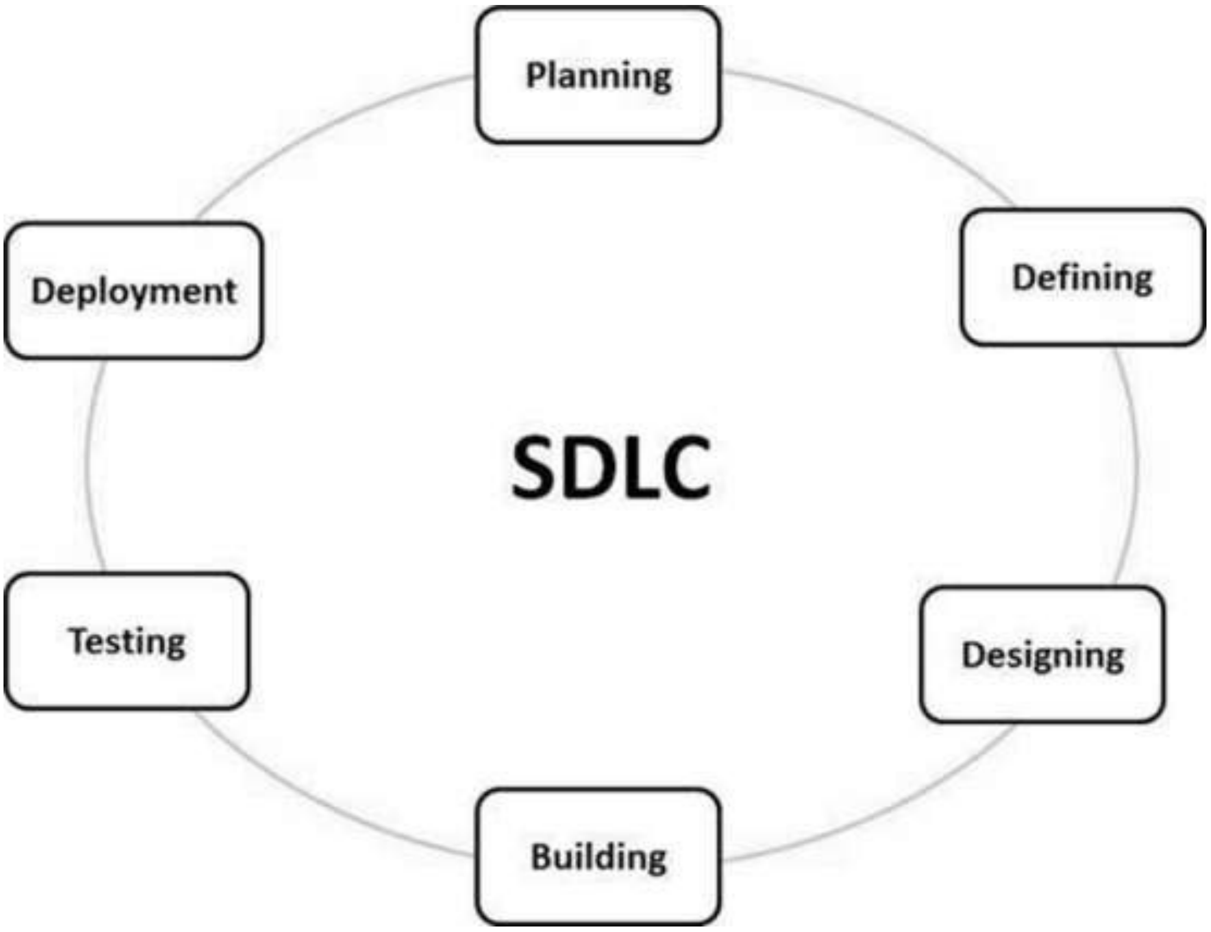
Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality softwares. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

- SDLC is the acronym of Software Development Life Cycle.
- It is also called as Software Development Process.
- SDLC is a framework defining tasks performed at each step in the software development process.
- ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

What is SDLC?

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development Life Cycle consists of the following stages −

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an **SRS (Software Requirement Specification)** document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT-User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

SDLC Models

There are various software development life cycle models defined and designed which are followed during the software development process. These models are also referred as Software Development Process Models". Each process model follows a Series of steps unique to its type to ensure success in the process of software development.

Following are the most important and popular SDLC models followed in the industry −

- Waterfall Model
- Iterative Model
- Spiral Model
- V-Model
- Big Bang Model

Other related methodologies are Agile Model, RAD Model, Rapid Application Development and Prototyping Models.

SDLC - Waterfall Model

The Waterfall Model was the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

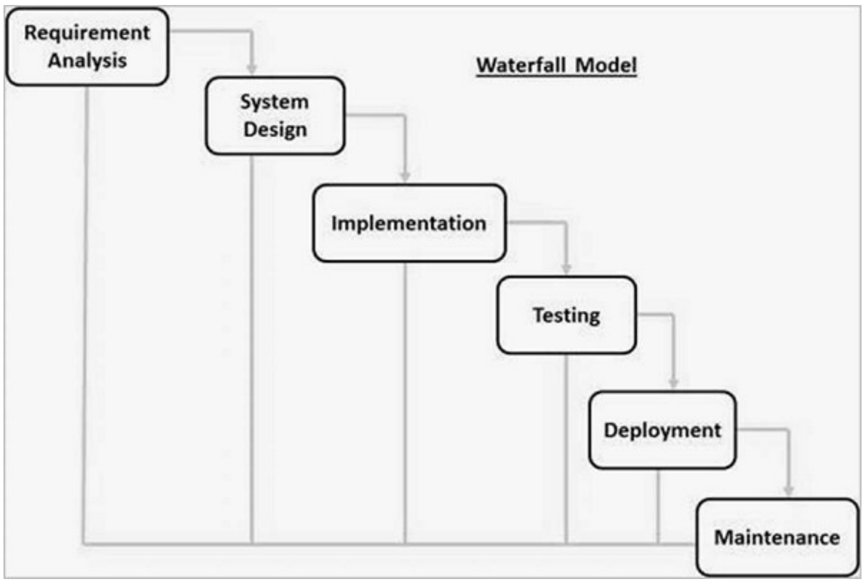
The Waterfall model is the earliest SDLC approach that was used for software development.

The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

Waterfall Model - Design

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are −

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.
- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model, phases do not overlap.

Waterfall Model - Application

Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are −

- Requirements are very well documented, clear and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available to support the product.
- The project is short.

Waterfall Model - Advantages

The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.

Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.

Some of the major advantages of the Waterfall Model are as follows −

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

Waterfall Model - Disadvantages

The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

The major disadvantages of the Waterfall Model are as follows −

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.
- Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

SDLC - Iterative Model

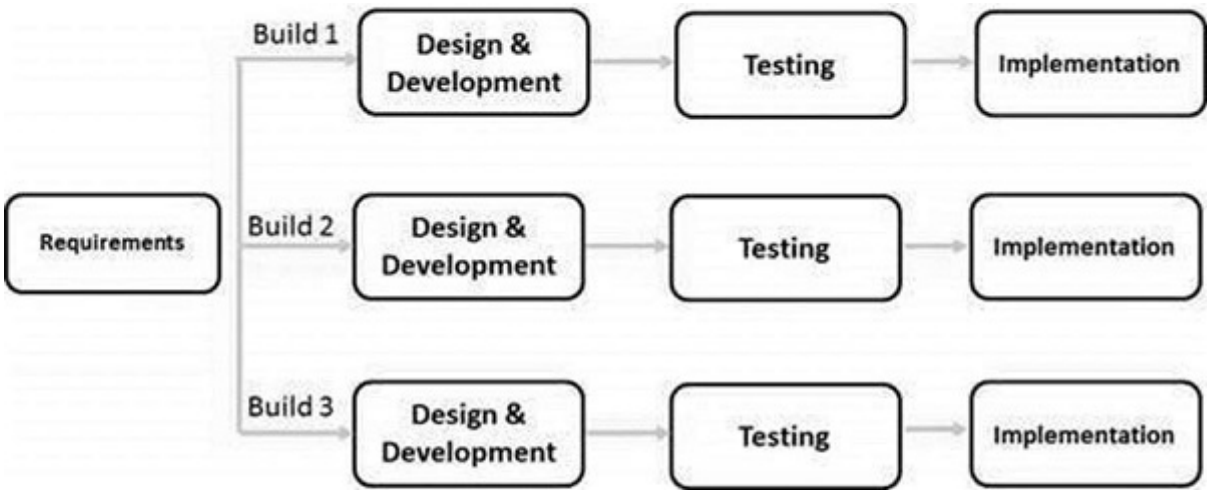
In the Iterative model, iterative process starts with a simple implementation of a small set of the software requirements and iteratively enhances the evolving versions until the complete system is implemented and ready to be deployed.

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which is then reviewed to identify further requirements. This process is then repeated, producing a new version of the software at the end of each iteration of the model.

Iterative Model - Design

Iterative process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added. The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental).

The following illustration is a representation of the Iterative and Incremental model −



Iterative and Incremental development is a combination of both iterative design or iterative method and incremental build model for development. "During software development, more than one iteration of the software development cycle may be in progress at the same time." This process may be described as an "evolutionary acquisition" or "incremental build" approach."

In this incremental model, the whole requirement is divided into various builds. During each iteration, the development module goes through the requirements, design, implementation and testing phases. Each subsequent release of the module adds function to the previous release. The process continues till the complete system is ready as per the requirement.

The key to a successful use of an iterative software development lifecycle is rigorous validation of requirements, and verification & testing of each version of the software against those requirements within each cycle of the model. As the software evolves through successive cycles, tests must be repeated and extended to verify each version of the software.

Iterative Model – Application

Like other SDLC models, Iterative and incremental development has some specific applications in the software industry. This model is most often used in the following scenarios −

- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some functionalities or requested enhancements may evolve with time.
- There is a time to the market constraint.
- A new technology is being used and is being learnt by the development team while working on the project.
- Resources with needed skill sets are not available and are planned to be used on contract basis for specific iterations.
- There are some high-risk features and goals which may change in the future.

Iterative Model - Pros and Cons

The advantage of this model is that there is a working model of the system at a very early stage of development, which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

The advantages of the Iterative and Incremental SDLC Model are as follows −

- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.

- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging during smaller iteration is easy.
- Risks are identified and resolved during iteration; and each iteration is an easily managed milestone.
- Easier to manage risk - High risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilized/applied to the next increment.
- Risk analysis is better.
- It supports changing requirements.
- Initial Operating time is less.
- Better suited for large and mission-critical projects.
- During the life cycle, software is produced early which facilitates customer evaluation and feedback.

The disadvantages of the Iterative and Incremental SDLC Model are as follows −

- More resources may be required.
- Although cost of change is lesser, but it is not very suitable for changing requirements.
- More management attention is required.
- System architecture or design issues may arise because not all requirements are gathered in the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.
- Not suitable for smaller projects.
- Management complexity is more.
- End of project may not be known which is a risk.
- Highly skilled resources are required for risk analysis.
- Projects progress is highly dependent upon the risk analysis phase.

SDLC - Spiral Model

The spiral model combines the idea of iterative development with the systematic, controlled aspects of the waterfall model. This Spiral model is a combination of iterative development process model and sequential linear development model i.e. the waterfall model with a very high emphasis on risk analysis. It allows incremental releases of the product or incremental refinement through each iteration around the spiral.

Spiral Model - Design

The spiral model has four phases. A software project repeatedly passes through these phases in iterations called Spirals.

Identification

This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase.

This phase also includes understanding the system requirements by continuous communication between the customer and the system analyst. At the end of the spiral, the product is deployed in the identified market.

Design

The Design phase starts with the conceptual design in the baseline spiral and involves architectural design, logical design of modules, physical product design and the final design in the subsequent spirals.

Construct or Build

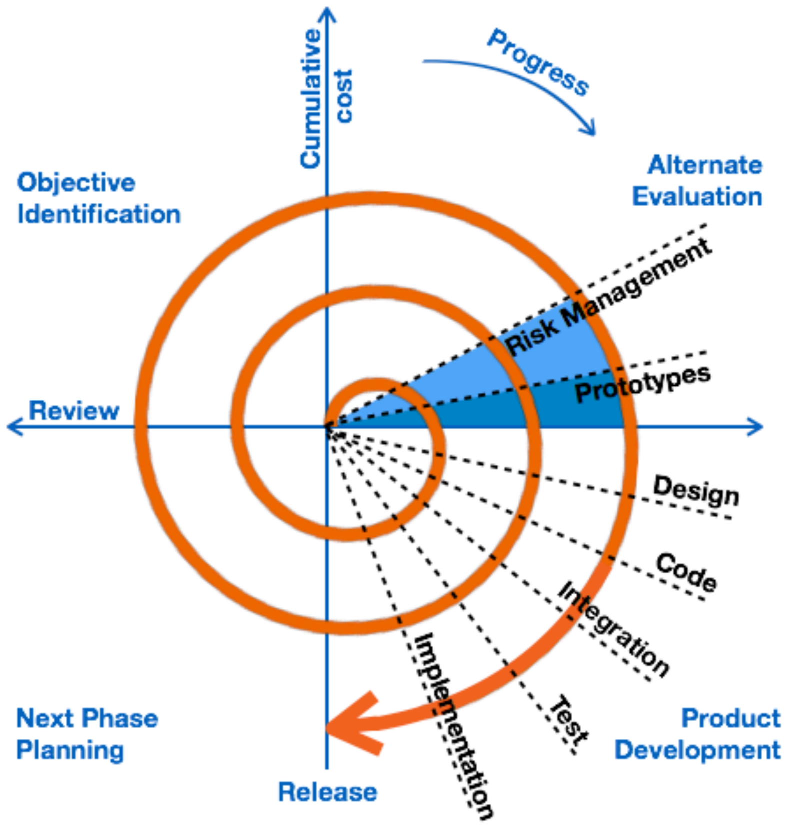
The Construct phase refers to production of the actual software product at every spiral. In the baseline spiral, when the product is just thought of and the design is being developed a POC (Proof of Concept) is developed in this phase to get customer feedback.

Then in the subsequent spirals with higher clarity on requirements and design details a working model of the software called build is produced with a version number. These builds are sent to the customer for feedback.

Evaluation and Risk Analysis

Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.

The following illustration is a representation of the Spiral Model, listing the activities in each phase.



Based on the customer evaluation, the software development process enters the next iteration and subsequently follows the linear approach to implement the feedback suggested by the customer. The process of iterations along the spiral continues throughout the life of the software.

Spiral Model Application

The Spiral Model is widely used in the software industry as it is in sync with the natural development process of any product, i.e. learning with maturity which involves minimum risk for the customer as well as the development firms.

The following pointers explain the typical uses of a Spiral Model −

- When there is a budget constraint and risk evaluation is important.
- For medium to high-risk projects.
- Long-term project commitment because of potential changes to economic priorities as the requirements change with time.
- Customer is not sure of their requirements which is usually the case.

- Requirements are complex and need evaluation to get clarity.
- New product line which should be released in phases to get enough customer feedback.
- Significant changes are expected in the product during the development cycle.

Spiral Model - Pros and Cons

The advantage of spiral lifecycle model is that it allows elements of the product to be added in, when they become available or known. This assures that there is no conflict with previous requirements and design.

This method is consistent with approaches that have multiple software builds and releases which allows making an orderly transition to a maintenance activity. Another positive aspect of this method is that the spiral model forces an early user involvement in the system development effort.

On the other side, it takes a very strict management to complete such products and there is a risk of running the spiral in an indefinite loop. So, the discipline of change and the extent of taking change requests is very important to develop and deploy the product successfully.

The advantages of the Spiral SDLC Model are as follows −

- Changing requirements can be accommodated.
- Allows extensive use of prototypes.
- Requirements can be captured more accurately.
- Users see the system early.
- Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.

The disadvantages of the Spiral SDLC Model are as follows −

- Management is more complex.
- End of the project may not be known early.
- Not suitable for small or low risk projects and could be expensive for small projects.
- Process is complex
- Spiral may go on indefinitely.
- Large number of intermediate stages requires excessive documentation.

SDLC - V-Model

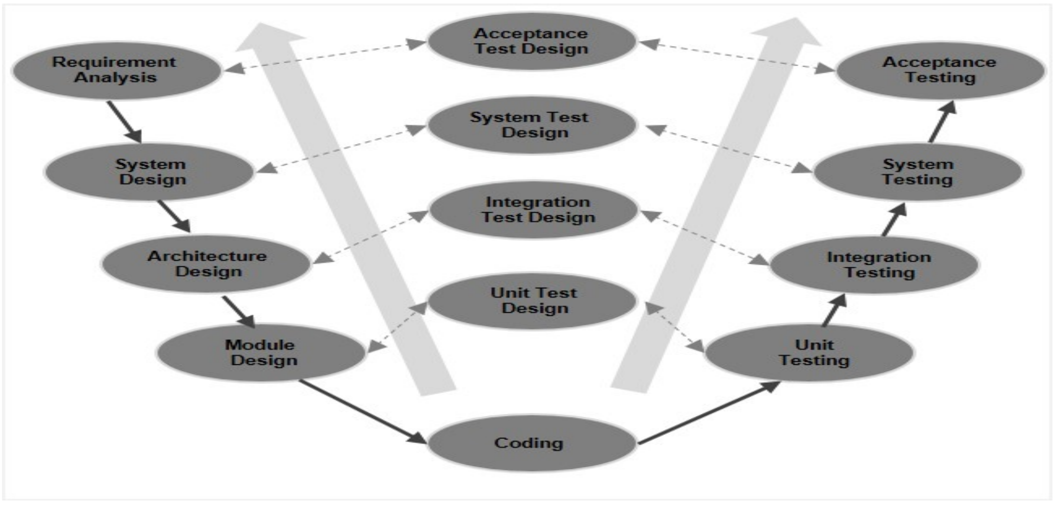
The V-model is an SDLC model where execution of processes happens in a sequential manner in a V-shape. It is also known as **Verification and Validation model**.

The V-Model is an extension of the waterfall model and is based on the association of a testing phase for each corresponding development stage. This means that for every single phase in the development cycle, there is a directly associated testing phase. This is a highly-disciplined model and the next phase starts only after completion of the previous phase.

V-Model - Design

Under the V-Model, the corresponding testing phase of the development phase is planned in parallel. So, there are Verification phases on one side of the 'V' and Validation phases on the other side. The Coding Phase joins the two sides of the V-Model.

The following illustration depicts the different phases in a V-Model of the SDLC.



V-Model - Verification Phases

There are several Verification phases in the V-Model, each of these are explained in detail below.

Business Requirement Analysis

This is the first phase in the development cycle where the product requirements are understood from the customer's perspective. This phase involves detailed communication with the customer to understand his expectations and exact requirement. This is a very important activity and needs to be managed well, as most of the customers are not sure about what exactly they need. The **acceptance test design planning** is done at this stage as business requirements can be used as an input for acceptance testing.

System Design

Once you have the clear and detailed product requirements, it is time to design the complete system. The system design will have the understanding and detailing the complete hardware and communication setup for the product under development. The system test plan is developed based on the system design. Doing this at an earlier stage leaves more time for the actual test execution later.

Architectural Design

Architectural specifications are understood and designed in this phase. Usually more than one technical approach is proposed and based on the technical and financial feasibility the final decision is taken. The system design is broken down further into modules taking up different functionality. This is also referred to as **High Level Design (HLD)**.

The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood and defined in this stage. With this information, integration tests can be designed and documented during this stage.

Module Design

In this phase, the detailed internal design for all the system modules is specified, referred to as **Low Level Design (LLD)**. It is important that the design is compatible with the other modules in the system architecture and the other external systems. The unit tests are an essential part of any development process and helps eliminate the maximum faults and errors at a very early stage. These unit tests can be designed at this stage based on the internal module designs.

Coding Phase

The actual coding of the system modules designed in the design phase is taken up in the Coding phase. The best suitable programming language is decided based on the system and architectural requirements.

The coding is performed based on the coding guidelines and standards. The code goes through numerous code reviews and is optimized for best performance before the final build is checked into the repository.

Validation Phases

The different Validation Phases in a V-Model are explained in detail below.

Unit Testing

Unit tests designed in the module design phase are executed on the code during this validation phase. Unit testing is the testing at code level and helps eliminate bugs at an early stage, though all defects cannot be uncovered by unit testing.

Integration Testing

Integration testing is associated with the architectural design phase. Integration tests are performed to test the coexistence and communication of the internal modules within the system.

System Testing

System testing is directly associated with the system design phase. System tests check the entire system functionality and the communication of the system under development with external systems. Most of the software and hardware compatibility issues can be uncovered during this system test execution.

Acceptance Testing

Acceptance testing is associated with the business requirement analysis phase and involves testing the product in user environment. Acceptance tests uncover the compatibility issues with the other systems available in the user environment. It also discovers the non-functional issues such as load and performance defects in the actual user environment.

V- Model ─ Application

V- Model application is almost the same as the waterfall model, as both the models are of sequential type. Requirements have to be very clear before the project starts, because it is usually expensive to go back and make changes. This model is used in the medical development field, as it is strictly a disciplined domain.

The following pointers are some of the most suitable scenarios to use the V-Model application.

- Requirements are well defined, clearly documented and fixed.
- Product definition is stable.
- Technology is not dynamic and is well understood by the project team.
- There are no ambiguous or undefined requirements.
- The project is short.

V-Model - Pros and Cons

The advantage of the V-Model method is that it is very easy to understand and apply. The simplicity of this model also makes it easier to manage. The disadvantage is that the model is not flexible to changes and just in case there is a requirement change, which is very common in today’s dynamic world, it becomes very expensive to make the change.

The advantages of the V-Model method are as follows −

- This is a highly-disciplined model and Phases are completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Simple and easy to understand and use.
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.

The disadvantages of the V-Model method are as follows −

- High risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.
- Once an application is in the testing stage, it is difficult to go back and change a functionality.
- No working software is produced until late during the life cycle.

SDLC - Big Bang Model

The Big Bang model is an SDLC model where we do not follow any specific process. The development just starts with the required money and efforts as the input, and the output is the software developed which may or may not be as per customer requirement. This Big Bang Model does not follow a process/procedure and there is a very little planning required. Even the customer is not sure about what exactly he wants and the requirements are implemented on the fly without much analysis.

Usually this model is followed for small projects where the development teams are very small.

Big Bang Model ─ Design and Application

The Big Bang Model comprises of focusing all the possible resources in the software development and coding, with very little or no planning. The requirements are understood and implemented as they come. Any changes required may or may not need to revamp the complete software.

This model is ideal for small projects with one or two developers working together and is also useful for academic or practice projects. It is an ideal model for the product where requirements are not well understood and the final release date is not given.

Big Bang Model - Pros and Cons

The advantage of this Big Bang Model is that it is very simple and requires very little or no planning. Easy to manage and no formal procedure are required.

However, the Big Bang Model is a very high risk model and changes in the requirements or misunderstood requirements may even lead to complete reversal or scraping of the project. It is ideal for repetitive or small projects with minimum risks.

The advantages of the Big Bang Model are as follows −

- This is a very simple model
- Little or no planning required
- Easy to manage
- Very few resources required
- Gives flexibility to developers
- It is a good learning aid for new comers or students.

The disadvantages of the Big Bang Model are as follows −

- Very High risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Can turn out to be very expensive if requirements are misunderstood.

SDLC - Agile Model

Agile SDLC model is a combination of iterative and incremental process models with focus on process adaptability and customer satisfaction by rapid delivery of working software product. Agile Methods break the product into small incremental builds. These builds are provided in iterations. Each iteration typically lasts from about one to three weeks. Every iteration involves cross functional teams working simultaneously on various areas like −

- Planning
- Requirements Analysis
- Design
- Coding
- Unit Testing and
- Acceptance Testing.

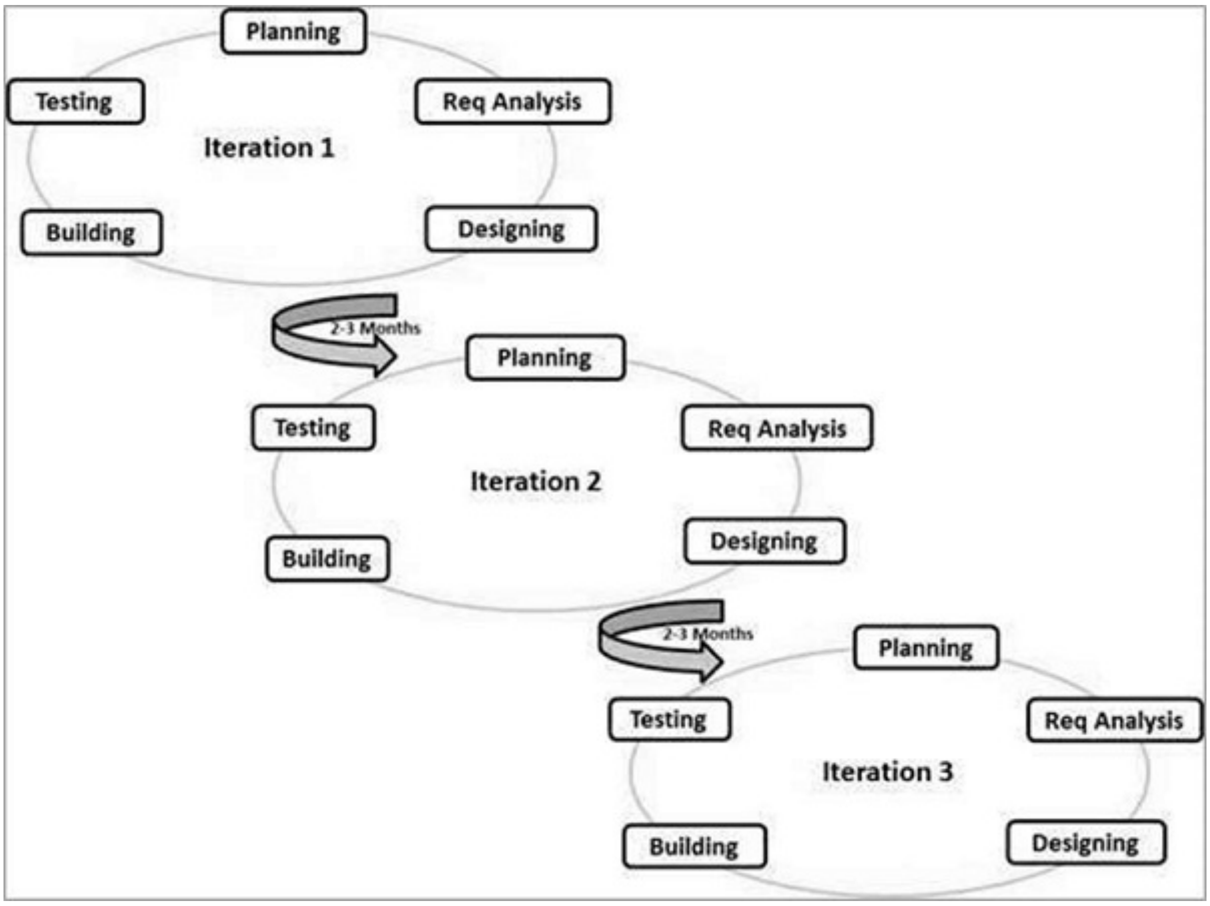
At the end of the iteration, a working product is displayed to the customer and important stakeholders.

What is Agile?

Agile model believes that every project needs to be handled differently and the existing methods need to be tailored to best suit the project requirements. In Agile, the tasks are divided to time boxes (small time frames) to deliver specific features for a release.

Iterative approach is taken and working software build is delivered after each iteration. Each build is incremental in terms of features; the final build holds all the features required by the customer.

Here is a graphical illustration of the Agile Model −



The Agile thought process had started early in the software development and started becoming popular with time due to its flexibility and adaptability.

The most popular Agile methods include Rational Unified Process (1994), Scrum (1995), Crystal Clear, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and

Dynamic Systems Development Method (DSDM) (1995). These are now collectively referred to as **Agile Methodologies**, after the Agile Manifesto was published in 2001.

Following are the Agile Manifesto principles −

- **Individuals and interactions** − In Agile development, self-organization and motivation are important, as are interactions like co-location and pair programming.
- **Working software** − Demo working software is considered the best means of communication with the customers to understand their requirements, instead of just depending on documentation.
- **Customer collaboration** − As the requirements cannot be gathered completely in the beginning of the project due to various factors, continuous customer interaction is very important to get proper product requirements.
- **Responding to change** − Agile Development is focused on quick responses to change and continuous development.

Agile Vs Traditional SDLC Models

Agile is based on the **adaptive software development methods**, whereas the traditional SDLC models like the waterfall model is based on a predictive approach. Predictive teams in the traditional SDLC models usually work with detailed planning and have a complete forecast of the exact tasks and features to be delivered in the next few months or during the product life cycle.

Predictive methods entirely depend on the **requirement analysis and planning** done in the beginning of cycle. Any changes to be incorporated go through a strict change control management and prioritization.

Agile uses an **adaptive approach** where there is no detailed planning and there is clarity on future tasks only in respect of what features need to be developed. There is feature driven development and the team adapts to the changing product requirements dynamically. The product is tested very frequently, through the release iterations, minimizing the risk of any major failures in future.

Customer Interaction is the backbone of this Agile methodology, and open communication with minimum documentation are the typical features of Agile development environment. The agile teams work in close collaboration with each other and are most often located in the same geographical location.

Agile Model - Pros and Cons

Agile methods are being widely accepted in the software world recently. However, this method may not always be suitable for all products. Here are some pros and cons of the Agile model.

The advantages of the Agile Model are as follows −

- Is a very realistic approach to software development.
- Promotes teamwork and cross training.
- Functionality can be developed rapidly and demonstrated.
- Resource requirements are minimum.
- Suitable for fixed or changing requirements
- Delivers early partial working solutions.
- Good model for environments that change steadily.
- Minimal rules, documentation easily employed.
- Enables concurrent development and delivery within an overall planned context.
- Little or no planning required.
- Easy to manage.
- Gives flexibility to developers.

The disadvantages of the Agile Model are as follows −

- Not suitable for handling complex dependencies.
- More risk of sustainability, maintainability and extensibility.
- An overall plan, an agile leader and agile PM practice is a must without which it will not work.
- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- There is a very high individual dependency, since there is minimum documentation generated.
- Transfer of technology to new team members may be quite challenging due to lack of documentation.

SDLC - RAD Model

The **RAD (Rapid Application Development)** model is based on prototyping and iterative development with no specific planning involved. The process of writing the software itself involves the planning required for developing the product.

Rapid Application Development focuses on gathering customer requirements through workshops or focus groups, early testing of the prototypes by the customer using iterative concept, reuse of the existing prototypes (components), continuous integration and rapid delivery.

What is RAD?

Rapid application development is a software development methodology that uses minimal planning in favor of rapid prototyping. A prototype is a working model that is functionally equivalent to a component of the product.

In the RAD model, the functional modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery. Since there is no detailed preplanning, it makes it easier to incorporate the changes within the development process.

RAD projects follow iterative and incremental model and have small teams comprising of developers, domain experts, customer representatives and other IT resources working progressively on their component or prototype.

The most important aspect for this model to be successful is to make sure that the prototypes developed are reusable.

RAD Model Design

RAD model distributes the analysis, design, build and test phases into a series of short, iterative development cycles.

Following are the various phases of the RAD Model −

Business Modelling

The business model for the product under development is designed in terms of flow of information and the distribution of information between various business channels. A complete business analysis is performed to find the vital information for business, how it can be obtained, how and when is the information processed and what are the factors driving successful flow of information.

Data Modelling

The information gathered in the Business Modelling phase is reviewed and analyzed to form sets of data objects vital for the business. The attributes of all data sets is identified and defined. The relation between these data objects are established and defined in detail in relevance to the business model.

Process Modelling

The data object sets defined in the Data Modelling phase are converted to establish the business information flow needed to achieve specific business objectives as per the business model. The process model for any changes or enhancements to the data object sets is defined in this phase. Process descriptions for adding, deleting, retrieving or modifying a data object are given.

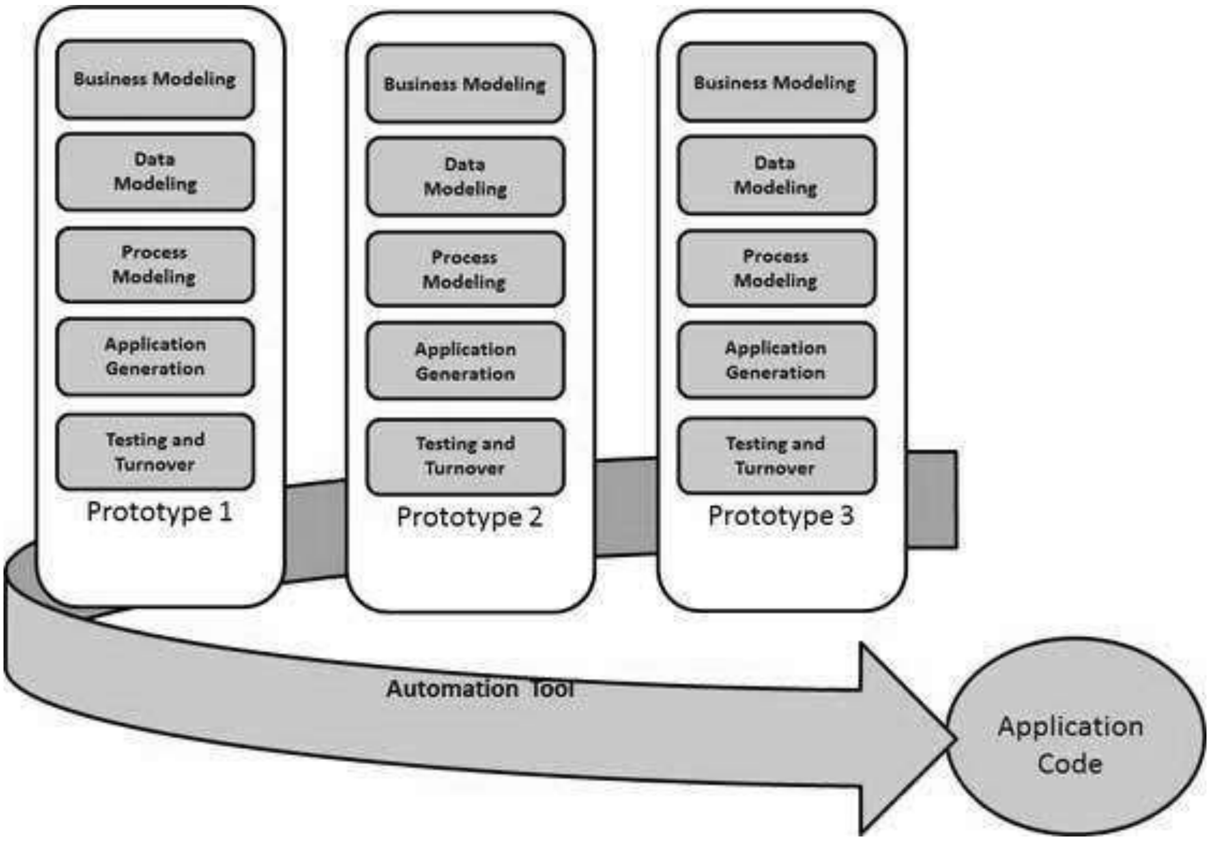
Application Generation

The actual system is built and coding is done by using automation tools to convert process and data models into actual prototypes.

Testing and Turnover

The overall testing time is reduced in the RAD model as the prototypes are independently tested during every iteration. However, the data flow and the interfaces between all the components need to be thoroughly tested with complete test coverage. Since most of the programming components have already been tested, it reduces the risk of any major issues.

The following illustration describes the RAD Model in detail.



RAD Model Vs Traditional SDLC

The traditional SDLC follows a rigid process models with high emphasis on requirement analysis and gathering before the coding starts. It puts pressure on the customer to sign off the requirements before the project starts and the customer doesn’t get the feel of the product as there is no working build available for a long time.

The customer may need some changes after he gets to see the software. However, the change process is quite rigid and it may not be feasible to incorporate major changes in the product in the traditional SDLC.

The RAD model focuses on iterative and incremental delivery of working models to the customer. This results in rapid delivery to the customer and customer involvement during the complete development cycle of product reducing the risk of non-conformance with the actual user requirements.

RAD Model - Application

RAD model can be applied successfully to the projects in which clear modularization is possible. If the project cannot be broken into modules, RAD may fail.

The following pointers describe the typical scenarios where RAD can be used −

- RAD should be used only when a system can be modularized to be delivered in an incremental manner.
- It should be used if there is a high availability of designers for Modelling.
- It should be used only if the budget permits use of automated code generating tools.
- RAD SDLC model should be chosen only if domain experts are available with relevant business knowledge.
- Should be used where the requirements change during the project and working prototypes are to be presented to customer in small iterations of 2-3 months.

RAD Model - Pros and Cons

RAD model enables rapid delivery as it reduces the overall development time due to the reusability of the components and parallel development. RAD works well only if high skilled engineers are available and the customer is also committed to achieve the targeted prototype in the given time frame. If there is commitment lacking on either side the model may fail.

The advantages of the RAD Model are as follows −

- Changing requirements can be accommodated.
- Progress can be measured.
- Iteration time can be short with use of powerful RAD tools.
- Productivity with fewer people in a short time.
- Reduced development time.
- Increases reusability of components.
- Quick initial reviews occur.
- Encourages customer feedback.
- Integration from very beginning solves a lot of integration issues.

The disadvantages of the RAD Model are as follows −

- Dependency on technically strong team members for identifying business requirements.
- Only system that can be modularized can be built using RAD.
- Requires highly skilled developers/designers.
- High dependency on Modelling skills.
- Inapplicable to cheaper projects as cost of Modelling and automated code generation is very high.
- Management complexity is more.
- Suitable for systems that are component based and scalable.
- Requires user involvement throughout the life cycle.
- Suitable for project requiring shorter development times.

SDLC - Software Prototype Model

The Software Prototyping refers to building software application prototypes which displays the functionality of the product under development, but may not actually hold the exact logic of the original software.

Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development. It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.

What is Software Prototyping?

Prototype is a working model of software with some limited functionality. The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.

Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Following is a stepwise approach explained to design a software prototype.

Basic Requirement Identification

This step involves understanding the very basics product requirements especially in terms of user interface. The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.

Developing the initial Prototype

The initial Prototype is developed in this stage, where the very basic requirements are showcased and user interfaces are provided. These features may not exactly work in the same manner internally in the actual software developed. While, the workarounds are used to give the same look and feel to the customer in the prototype developed.

Review of the Prototype

The prototype developed is then presented to the customer and the other important stakeholders in the project. The feedback is collected in an organized manner and used for further enhancements in the product under development.

Revise and Enhance the Prototype

The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like – time and budget constraints and technical feasibility of the actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until the customer expectations are met.

Prototypes can have horizontal or vertical dimensions. A Horizontal prototype displays the user interface for the product and gives a broader view of the entire system, without concentrating on internal functions. A Vertical prototype on the other side is a detailed elaboration of a specific function or a sub system in the product.

The purpose of both horizontal and vertical prototype is different. Horizontal prototypes are used to get more information on the user interface level and the business requirements. It can even be presented in the sales demos to get business in the market. Vertical prototypes are technical in nature and are used to get details of the exact functioning of the sub systems. For example, database requirements, interaction and data processing loads in a given sub system.

Software Prototyping - Types

There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely −

Throwaway/Rapid Prototyping

Throwaway prototyping is also called as rapid or close ended prototyping. This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype. Once the actual requirements are understood, the prototype is discarded and the actual system is developed with a much clear understanding of user requirements.

Evolutionary Prototyping

Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning. The prototype developed forms the heart of the future prototypes on top of which the entire system is built. By using evolutionary prototyping, the well-understood requirements are included in the prototype and the requirements are added as and when they are understood.

Incremental Prototyping

Incremental prototyping refers to building multiple functional prototypes of the various sub-systems and then integrating all the available prototypes to form a complete system.

Extreme Prototyping

Extreme prototyping is used in the web development domain. It consists of three sequential phases. First, a basic prototype with all the existing pages is presented in the HTML format. Then the data processing is simulated using a prototype services layer. Finally, the services are implemented and integrated to the final prototype. This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.

Software Prototyping - Application

Software Prototyping is most useful in development of systems having high level of user interactions such as online systems. Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed.

Software that involves too much of data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping. Prototype development could be an extra overhead in such projects and may need lot of extra efforts.

Software Prototyping - Pros and Cons

Software prototyping is used in typical cases and the decision should be taken very carefully so that the efforts spent in building the prototype add considerable value to the final software developed. The model has its own pros and cons discussed as follows.

The advantages of the Prototyping Model are as follows −

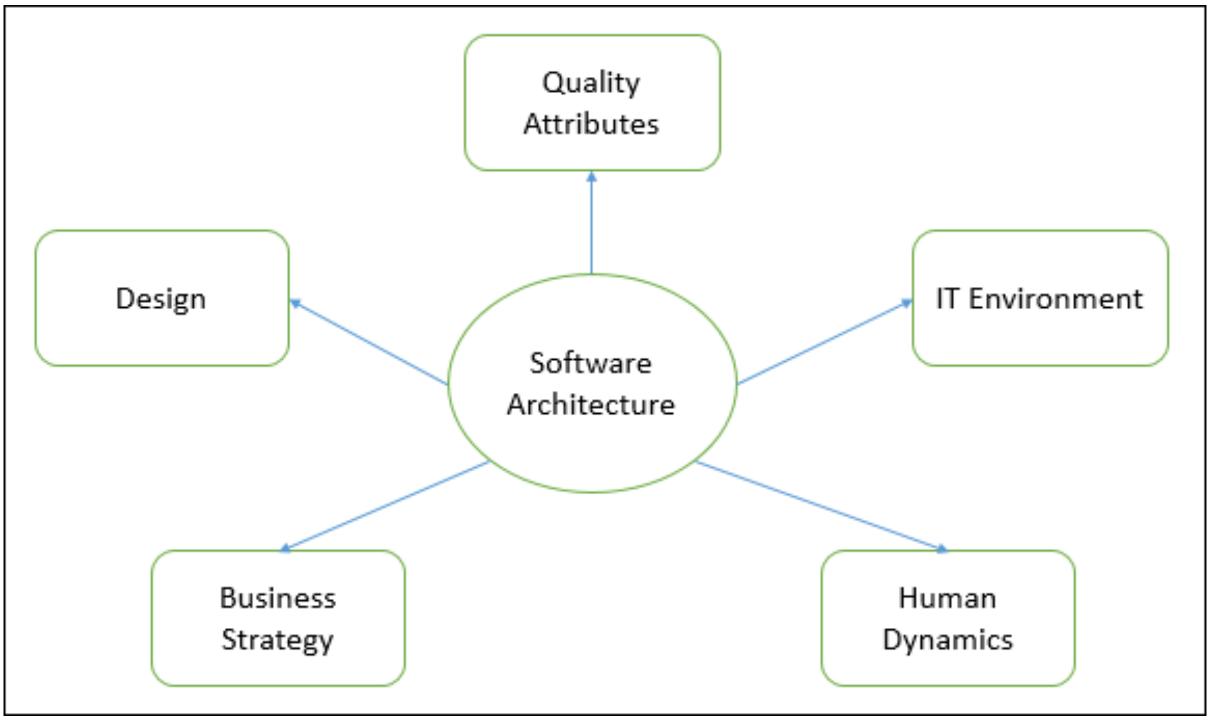
- Increased user involvement in the product even before its implementation.
- Since a working model of the system is displayed, the users get a better understanding of the system being developed.
- Reduces time and cost as the defects can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily.
- Confusing or difficult functions can be identified.

The Disadvantages of the Prototyping Model are as follows −

- Risk of insufficient requirement analysis owing to too much dependency on the prototype.
- Users may get confused in the prototypes and actual systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Developers may try to reuse the existing prototypes to build the actual system, even when it is not technically feasible.
- The effort invested in building prototypes may be too much if it is not monitored properly.

Software Architecture & Design Introduction

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In **Architecture**, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

Software Architecture

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

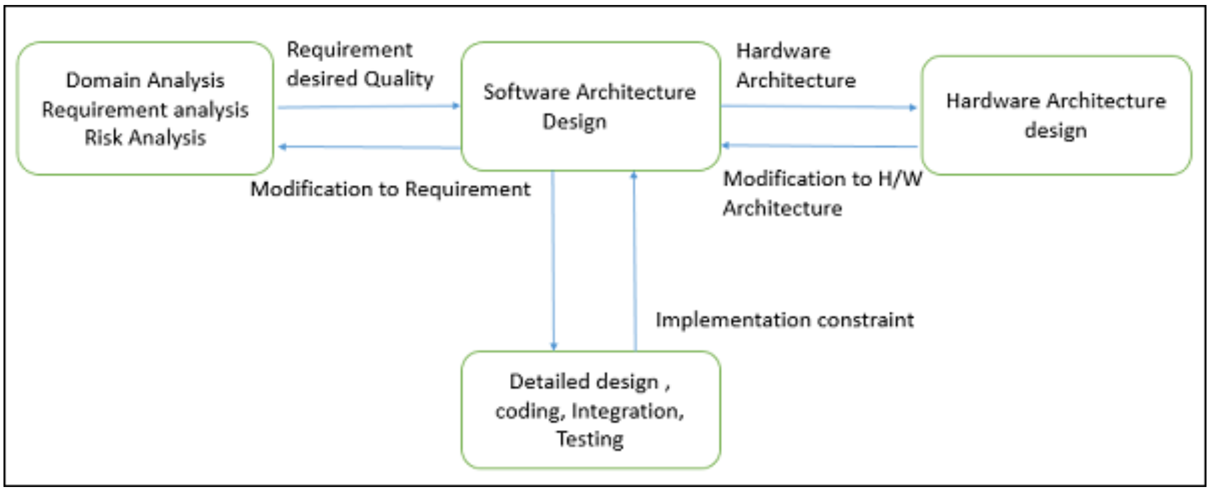
- It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
- Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of −
  - Selection of structural elements and their interfaces by which the system is composed.
  - Behavior as specified in collaborations among those elements.
  - Composition of these structural and behavioral elements into large subsystem.
  - Architectural decisions align with business objectives.
  - Architectural styles guide the organization.

Software Design

Software design provides a **design plan** that describes the elements of a system, how they fit, and work together to fulfill the requirement of the system. The objectives of having a design plan are as follows –

- To negotiate system requirements, and to set expectations with customers, marketing, and management personnel.
- Act as a blueprint during the development process.
- Guide the implementation tasks, including detailed design, coding, integration, and testing.

It comes before the detailed design, coding, integration, and testing and after the domain analysis, requirements analysis, and risk analysis.


Domain Analysis
Requirement analysis
Risk Analysis
Requirement
desired Quality
Modification to Requirement
Software Architecture Design
Hardware Architecture
Modification to H/W Architecture
Hardware Architecture design
Implementation constraint
Detailed design , coding, Integration, Testing

Goals of Architecture

The primary goal of the architecture is to identify requirements that affect the structure of the application. A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.

Some of the other goals are as follows –

- Expose the structure of the system, but hide its implementation details.
- Realize all the use-cases and scenarios.
- Try to address the requirements of various stakeholders.
- Handle both functional and quality requirements.
- Reduce the goal of ownership and improve the organization’s market position.
- Improve quality and functionality offered by the system.
- Improve external confidence in either the organization or system.

Limitations

Software architecture is still an emerging discipline within software engineering. It has the following limitations –

- Lack of tools and standardized ways to represent architecture.
- Lack of analysis methods to predict whether architecture will result in an implementation that meets the requirements.
- Lack of awareness of the importance of architectural design to software development.
- Lack of understanding of the role of software architect and poor communication among stakeholders.
- Lack of understanding of the design process, design experience and evaluation of design.

Role of Software Architect

A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas −

Design Expertise

- Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.
- Lead the development team and coordinate the development efforts for the integrity of the design.
- Should be able to review design proposals and tradeoff among themselves.

Domain Expertise

- Expert on the system being developed and plan for software evolution.
- Assist in the requirement investigation process, assuring completeness and consistency.
- Coordinate the definition of domain model for the system being developed.

Technology Expertise

- Expert on available technologies that helps in the implementation of the system.
- Coordinate the selection of programming language, framework, platforms, databases, etc.

Methodological Expertise

- Expert on software development methodologies that may be adopted during SDLC (Software Development Life Cycle).
- Choose the appropriate approaches for development that helps the entire team.

Hidden Role of Software Architect

- Facilitates the technical work among team members and reinforcing the trust relationship in the team.
- Information specialist who shares knowledge and has vast experience.
- Protect the team members from external forces that would distract them and bring less value to the project.

Deliverables of the Architect

- A clear, complete, consistent, and achievable set of functional goals
- A functional description of the system, with at least two layers of decomposition
- A concept for the system
- A design in the form of the system, with at least two layers of decomposition
- A notion of the timing, operator attributes, and the implementation and operation plans
- A document or process which ensures functional decomposition is followed, and the form of interfaces is controlled

Quality Attributes

Quality is a measure of excellence or the state of being free from deficiencies or defects. Quality attributes are the system properties that are separate from the functionality of the system.

Implementing quality attributes makes it easier to differentiate a good system from a bad one. Attributes are overall factors that affect runtime behavior, system design, and user experience.

They can be classified as −

Static Quality Attributes

Reflect the structure of a system and organization, directly related to architecture, design, and source code. They are invisible to end-user, but affect the development and maintenance cost, e.g.: modularity, testability, maintainability, etc.

Dynamic Quality Attributes

Reflect the behavior of the system during its execution. They are directly related to system's architecture, design, source code, configuration, deployment parameters, environment, and platform.

They are visible to the end-user and exist at runtime, e.g. throughput, robustness, scalability, etc.

Quality Scenarios

Quality scenarios specify how to prevent a fault from becoming a failure. They can be divided into six parts based on their attribute specifications −

- **Source** − An internal or external entity such as people, hardware, software, or physical infrastructure that generate the stimulus.
- **Stimulus** − A condition that needs to be considered when it arrives on a system.
- **Environment** − The stimulus occurs within certain conditions.
- **Artifact** − A whole system or some part of it such as processors, communication channels, persistent storage, processes etc.
- **Response** − An activity undertaken after the arrival of stimulus such as detect faults, recover from fault, disable event source etc.
- **Response measure** − Should measure the occurred responses so that the requirements can be tested.

Common Quality Attributes

The following table lists the common quality attributes a software architecture must have −

Category	Quality Attribute	Description
Design Qualities	Conceptual Integrity	Defines the consistency and coherence of the overall design. This includes the way components or modules are designed.
	Maintainability	Ability of the system to undergo changes with a degree of ease.
	Reusability	Defines the capability for components and subsystems to be suitable for use in other applications.
Run-time Qualities	Interoperability	Ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties.
	Manageability	Defines how easy it is for system administrators to manage the application.
	Reliability	Ability of a system to remain operational over time.
	Scalability	Ability of a system to either handle the load increase without impacting the performance of the system or the ability to be readily enlarged.
	Security	Capability of a system to prevent malicious or accidental actions outside of the designed usages.
	Performance	Indication of the responsiveness of a system to execute any action within a given time interval.
	Availability	Defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period.

System Qualities	Supportability	Ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly.
	Testability	Measure of how easy it is to create test criteria for the system and its components.
User Qualities	Usability	Defines how well the application meets the requirements of the user and consumer by being intuitive.
Architecture Quality	Correctness	Accountability for satisfying all the requirements of the system.
Non-runtime Quality	Portability	Ability of the system to run under different computing environment.
	Integrality	Ability to make separately developed components of the system work correctly together.
	Modifiability	Ease with which each software system can accommodate changes to its software.
Business quality attributes	Cost and schedule	Cost of the system with respect to time to market, expected project lifetime & utilization of legacy.
	Marketability	Use of system with respect to market competition.

Key Principles

Software architecture is described as the organization of a system, where the system represents a set of components that accomplish the defined functions.

Architectural Style

The **architectural style**, also called as **architectural pattern**, is a set of principles which shapes an application. It defines an abstract framework for a family of system in terms of the pattern of structural organization.

The architectural style is responsible to −

- Provide a lexicon of components and connectors with rules on how they can be combined.
- Improve partitioning and allow the reuse of design by giving solutions to frequently occurring problems.
- Describe a particular way to configure a collection of components (a module with well-defined interfaces, reusable, and replaceable) and connectors (communication link between modules).

The software that is built for computer-based systems exhibit one of many architectural styles. Each style describes a system category that encompasses −

- A set of component types which perform a required function by the system.
- A set of connectors (subroutine call, remote procedure call, data stream, and socket) that enable communication, coordination, and cooperation among different components.
- Semantic constraints which define how components can be integrated to form the system.
- A topological layout of the components indicating their runtime interrelationships.

Common Architectural Design

The following table lists architectural styles that can be organized by their key focus area −

Category	Architectural Design	Description
Communication	Message bus	Prescribes use of a software system that can receive and send messages using one or more communication channels.
	Service–Oriented Architecture (SOA)	Defines the applications that expose and consume functionality as a service using contracts and messages.
Deployment	Client/server	Separate the system into two applications, where the client makes requests to the server.
	3-tier or N-tier	Separates the functionality into separate segments with each segment being a tier located on a physically separate computer.
Domain	Domain Driven Design	Focused on modeling a business domain and defining business objects based on entities within the business domain.
Structure	Component Based	Breakdown the application design into reusable functional or logical components that expose well-defined communication interfaces.
	Layered	Divide the concerns of the application into stacked groups (layers).

	Object oriented	Based on the division of responsibilities of an application or system into objects, each containing the data and the behavior relevant to the object.

Types of Architecture

There are four types of architecture from the viewpoint of an enterprise and collectively, these architectures are referred to as **enterprise architecture**.

- **Business architecture** − Defines the strategy of business, governance, organization, and key business processes within an enterprise and focuses on the analysis and design of business processes.
- **Application (software) architecture** − Serves as the blueprint for individual application systems, their interactions, and their relationships to the business processes of the organization.
- **Information architecture** − Defines the logical and physical data assets and data management resources.
- **Information technology (IT) architecture** − Defines the hardware and software building blocks that make up the overall information system of the organization.

Architecture Design Process

The architecture design process focuses on the decomposition of a system into different components and their interactions to satisfy functional and nonfunctional requirements. The key inputs to software architecture design are −

- The requirements produced by the analysis tasks.
- The hardware architecture (the software architect in turn provides requirements to the system architect, who configures the hardware architecture).

The result or output of the architecture design process is an **architectural description**. The basic architecture design process is composed of the following steps −

Understand the Problem

- This is the most crucial step because it affects the quality of the design that follows.
- Without a clear understanding of the problem, it is not possible to create an effective solution.
- Many software projects and products are considered failures because they did not actually solve a valid business problem or have a recognizable return on investment (ROI).

Identify Design Elements and their Relationships

- In this phase, build a baseline for defining the boundaries and context of the system.
- Decomposition of the system into its main components based on functional requirements. The decomposition can be modeled using a design structure matrix (DSM), which shows the dependencies between design elements without specifying the granularity of the elements.
- In this step, the first validation of the architecture is done by describing a number of system instances and this step is referred as functionality based architectural design.

Evaluate the Architecture Design

- Each quality attribute is given an estimate so in order to gather qualitative measures or quantitative data, the design is evaluated.
- It involves evaluating the architecture for conformance to architectural quality attributes requirements.
- If all estimated quality attributes are as per the required standard, the architectural design process is finished.
- If not, the third phase of software architecture design is entered: architecture transformation. If the observed quality attribute does not meet its requirements, then a new design must be created.

Transform the Architecture Design

- This step is performed after an evaluation of the architectural design. The architectural design must be changed until it completely satisfies the quality attribute requirements.
- It is concerned with selecting design solutions to improve the quality attributes while preserving the domain functionality.
- A design is transformed by applying design operators, styles, or patterns. For transformation, take the existing design and apply design operator such as decomposition, replication, compression, abstraction, and resource sharing.
- The design is again evaluated and the same process is repeated multiple times if necessary and even performed recursively.
- The transformations (i.e. quality attribute optimizing solutions) generally improve one or some quality attributes while they affect others negatively

Key Architecture Principles

Following are the key principles to be considered while designing an architecture −

Build to Change Instead of Building to Last

Consider how the application may need to change over time to address new requirements and challenges, and build in the flexibility to support this.

Reduce Risk and Model to Analyze

Use design tools, visualizations, modeling systems such as UML to capture requirements and design decisions. The impacts can also be analyzed. Do not formalize the model to the extent that it suppresses the capability to iterate and adapt the design easily.

Use Models and Visualizations as a Communication and Collaboration Tool

Efficient communication of the design, the decisions, and ongoing changes to the design is critical to good architecture. Use models, views, and other visualizations of the architecture to communicate and share the design efficiently with all the stakeholders. This enables rapid communication of changes to the design.

Identify and understand key engineering decisions and areas where mistakes are most often made. Invest in getting key decisions right the first time to make the design more flexible and less likely to be broken by changes.

Use an Incremental and Iterative Approach

Start with baseline architecture and then evolve candidate architectures by iterative testing to improve the architecture. Iteratively add details to the design over multiple passes to get the big or right picture and then focus on the details.

Key Design Principles

Following are the design principles to be considered for minimizing cost, maintenance requirements, and maximizing extendibility, usability of architecture −

Separation of Concerns

Divide the components of system into specific features so that there is no overlapping among the components functionality. This will provide high cohesion and low coupling. This approach avoids the interdependency among components of system which helps in maintaining the system easy.

Single Responsibility Principle

Each and every module of a system should have one specific responsibility, which helps the user to clearly understand the system. It should also help with integration of the component with other components.

Principle of Least Knowledge

Any component or object should not have the knowledge about internal details of other components. This approach avoids interdependency and helps maintainability.

Minimize Large Design Upfront

Minimize large design upfront if the requirements of an application are unclear. If there is a possibility of modifying requirements, then avoid making a large design for whole system.

Do not Repeat the Functionality

Do not repeat functionality specifies that functionality of components should not to be repeated and hence a piece of code should be implemented in one component only. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.

Prefer Composition over Inheritance while Reusing the Functionality

Inheritance creates dependency between children and parent classes and hence it blocks the free use of the child classes. In contrast, the composition provides a great level of freedom and reduces the inheritance hierarchies.

Identify Components and Group them in Logical Layers

Identity components and the area of concern that are needed in system to satisfy the requirements. Then group these related components in a logical layer, which will help the user to understand the structure of the system at a high level. Avoid mixing components of different type of concerns in same layer.

Define the Communication Protocol between Layers

Understand how components will communicate with each other which requires a complete knowledge of deployment scenarios and the production environment.

Define Data Format for a Layer

Various components will interact with each other through data format. Do not mix the data formats so that applications are easy to implement, extend, and maintain. Try to keep data format same for a layer, so that various components need not code/decode the data while communicating with each other. It reduces a processing overhead.

System Service Components should be Abstract

Code related to security, communications, or system services like logging, profiling, and configuration should be abstracted in the separate components. Do not mix this code with business logic, as it is easy to extend design and maintain it.

Design Exceptions and Exception Handling Mechanism

Defining exceptions in advance, helps the components to manage errors or unwanted situation in an elegant manner. The exception management will be same throughout the system.

Naming Conventions

Naming conventions should be defined in advance. They provide a consistent model that helps the users to understand the system easily. It is easier for team members to validate code written by others, and hence will increase the maintainability.

Architecture Models

Software architecture involves the high level structure of software system abstraction, by using decomposition and composition, with architectural style and quality attributes. A software architecture design must conform to the major functionality and performance requirements of the system, as well as satisfy the non-functional requirements such as reliability, scalability, portability, and availability.

A software architecture must describe its group of components, their connections, interactions among them and deployment configuration of all components.

A software architecture can be defined in many ways −

- **UML (Unified Modeling Language)** − UML is one of object-oriented solutions used in software modeling and design.
- **Architecture View Model (4+1 view model)** − Architecture view model represents the functional and non-functional requirements of software application.
- **ADL (Architecture Description Language)** − ADL defines the software architecture formally and semantically.

Component-Based Architecture

Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based architecture is to ensure **component reusability**. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit. There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services. These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation.

Component-oriented software design has many advantages over the traditional object-oriented approaches such as −

- Reduced time in market and the development cost by reusing existing components.
- Increased reliability with the reuse of the existing components.

What is a Component?

A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.

A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities. It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture.

A software component can be defined as a unit of composition with a contractually specified interface and explicit context dependencies only. That is, a software component can be deployed independently and is subject to composition by third parties.

Views of a Component

A component can have three different views − object-oriented view, conventional view, and process-related view.

Object-oriented view

A component is viewed as a set of one or more cooperating classes. Each problem domain class (analysis) and infrastructure class (design) are explained to identify all attributes and operations that apply to its implementation. It also involves defining the interfaces that enable classes to communicate and cooperate.

Conventional view

It is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it.

Process-related view

In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library. As the software architecture is formulated, components are selected from the library and used to populate the architecture.

- A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.
- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach.
- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean (EJB), .NET components, and CORBA components.

Characteristics of Components

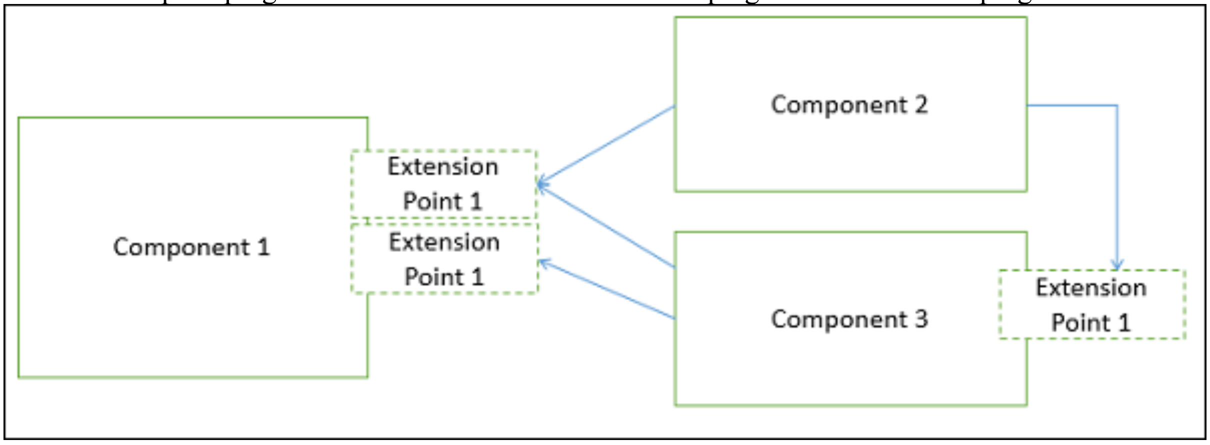
- **Reusability** − Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task.
- **Replaceable** − Components may be freely substituted with other similar components.
- **Not context specific** − Components are designed to operate in different environments and contexts.
- **Extensible** − A component can be extended from existing components to provide new behavior.
- **Encapsulated** − A A component depicts the interfaces, which allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state.
- **Independent** − Components are designed to have minimal dependencies on other components.

Principles of Component−Based Design

A component-level design can be represented by using some intermediary representation (e.g. graphical, tabular, or text-based) that can be translated into source code. The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors.

- The software system is decomposed into reusable, cohesive, and encapsulated component units.
- Each component has its own interface that specifies required ports and provided ports; each component hides its detailed implementation.
- A component should be extended without the need to make internal code or design modifications to the existing parts of the component.
- Depend on abstractions component do not depend on other concrete components, which increase difficulty in expendability.
- Connectors connected components, specifying and ruling the interaction among components. The interaction type is specified by the interfaces of the components.

- Components interaction can take the form of method invocations, asynchronous invocations, broadcasting, message driven interactions, data stream communications, and other protocol specific interactions.
- For a server class, specialized interfaces should be created to serve major categories of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface.
- A component can extend to other components and still offer its own extension points. It is the concept of plug-in based architecture. This allows a plugin to offer another plugin API.



Component-Level Design Guidelines

Creates a naming conventions for components that are specified as part of the architectural model and then refines or elaborates as part of the component-level model.

- Attains architectural component names from the problem domain and ensures that they have meaning to all stakeholders who view the architectural model.
- Extracts the business process entities that can exist independently without any associated dependency on other entities.
- Recognizes and discover these independent entities as new components.
- Uses infrastructure component names that reflect their implementation-specific meaning.
- Models any dependencies from left to right and inheritance from top (base class) to bottom (derived classes).
- Model any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

Conducting Component-Level Design

Recognizes all design classes that correspond to the problem domain as defined in the analysis model and architectural model.

- Recognizes all design classes that correspond to the infrastructure domain.
- Describes all design classes that are not acquired as reusable components, and specifies message details.
- Identifies appropriate interfaces for each component and elaborates attributes and defines data types and data structures required to implement them.
- Describes processing flow within each operation in detail by means of pseudo code or UML activity diagrams.
- Describes persistent data sources (databases and files) and identifies the classes required to manage them.
- Develop and elaborates behavioral representations for a class or component. This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class.
- Elaborates deployment diagrams to provide additional implementation detail.

- Demonstrates the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environment.
- The final decision can be made by using established design principles and guidelines. Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model.

Advantages

- **Ease of deployment** − As new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole.
- **Reduced cost** − The use of third-party components allows you to spread the cost of development and maintenance.
- **Ease of development** − Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- **Reusable** − The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- **Modification of technical complexity** − A component modifies the complexity through the use of a component container and its services.
- **Reliability** − The overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse.
- **System maintenance and evolution** − Easy to change and update the implementation without affecting the rest of the system.
- **Independent** − Independency and flexible connectivity of components. Independent development of components by different group in parallel. Productivity for the software development and future software development.

What are Connectors?

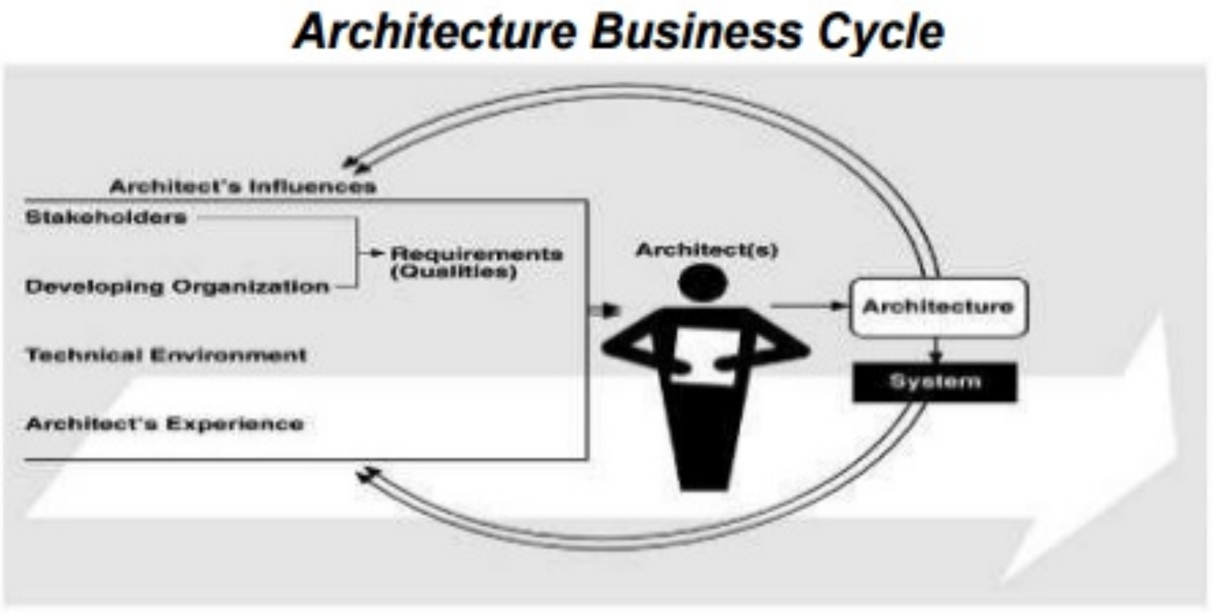
The software that is built for computer-based systems exhibit one of many architectural styles. Each style describes a system category that encompasses −

- A set of component types which perform a required function by the system.
- A set of connectors (subroutine call, remote procedure call, data stream, and socket) that enable communication, coordination, and cooperation among different components.
- Semantic constraints which define how components can be integrated to form the system.
- A topological layout of the components indicating their runtime interrelationships.

The Architecture Business Cycle:

Definition: Architecture Business Cycle (ABC):

"Software architecture is a result of technical, business, and social influences. Its existence in turn affects the technical, business, and social environments that subsequently influence future architectures. We call this cycle of influences, from the environment to the architecture and back to the environment, the Architecture Business Cycle (ABC)."



1.The organization goals of **Architecture Business Cycle** are beget requirements, which beget an architecture, which begets a system. The architecture flows from the architect's experience and the technical environment of the day.

2.Three things required for ABC are as follows:

i. Case studies of successful architectures crafted to satisfy demanding requirements, so as to help set the technical playing field of the day.

ii. Methods to assess an architecture before any system is built from it, so as to mitigate the risks associated with launching unprecedented designs. **iii.Techniques** for incremental architecture-based development, so as to uncover design flaws before it is too late to correct them.

How the ABC Works :

1. The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system; as we will see, it particularly prescribes the units of software that must be implemented (or otherwise obtained) and integrated to form the system. These units are the basis for the development project's structure. Teams are formed for individual software units; and the development, test, and integration activities all revolve around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. If a company becomes adept at building families of similar systems, it will tend to invest in each team by nurturing each area of expertise. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization.

In the software product line case study, separate groups were given responsibility for building and maintaining individual portions of the organization's architecture for a family of products. In any design undertaken by the organization at large, these groups have a strong voice in the system's decomposition, pressuring for the continued existence of the portions they control.

2. The architecture can affect the goals of the developing organization. A successful system built from it can enable a company to establish a foothold in a particular market area. The architecture can provide opportunities for the efficient

production and deployment of similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.

3. The architecture can affect customer requirements for the next system by giving the customer the opportunity to receive a system (based on the same architecture) in a more reliable, timely, and economical manner than if the subsequent system were to be built from scratch. The customer may be willing to relax some requirements to gain these economies. Shrink-wrapped software has clearly affected people's requirements by providing solutions that are not tailored to their precise needs but are instead inexpensive and (in the best of all possible worlds) of high quality. Product lines have the same effect on customers who cannot be so flexible with their requirements. A Case Study in Product Line Development will show how a product line architecture caused customers to happily compromise their requirements because they could get high-quality software that fit their basic needs quickly, reliably, and at lower cost.

4. The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base. A system that was successfully built around a tool bus or .NET or encapsulated finite-state machines will engender similar systems built the same way in the future. On the other hand, architectures that fail are less likely to be chosen for future projects.

5. A few systems will influence and actually change the software engineering culture, that is, the technical environment in which system builders operate and learn. The first relational databases, compiler generators, and table-driven operating systems had this effect in the 1960s and early 1970s; the first spreadsheets and windowing systems, in the 1980s. The World Wide Web is the example for

the 1990s. J2EE may be the example for the first decade of the twenty-first century. When such pathfinder systems are constructed, subsequent systems are affected by their legacy.

These and other feedback mechanisms form what we call the ABC, illustrated in Figure , which depicts the influences of the culture and business of the development organization on the software architecture. That architecture is, in turn, a primary determinant of the properties of the developed system or systems. But the ABC is also based on a recognition that shrewd organizations can take advantage of the organizational and experiential effects of developing an architecture and can use those effects to position their business strategically for future projects.

Building the ABC:

Building the ABC is done by identifying the influences to and from architectures as follows:

ARCHITECTURES ARE INFLUENCED BY SYSTEM STAKEHOLDERS:

1. Many people and organizations are interested in the construction of a software system.

2. Stakeholders are:

· The customer,

· the end users,

· the developers,

· the project manager,

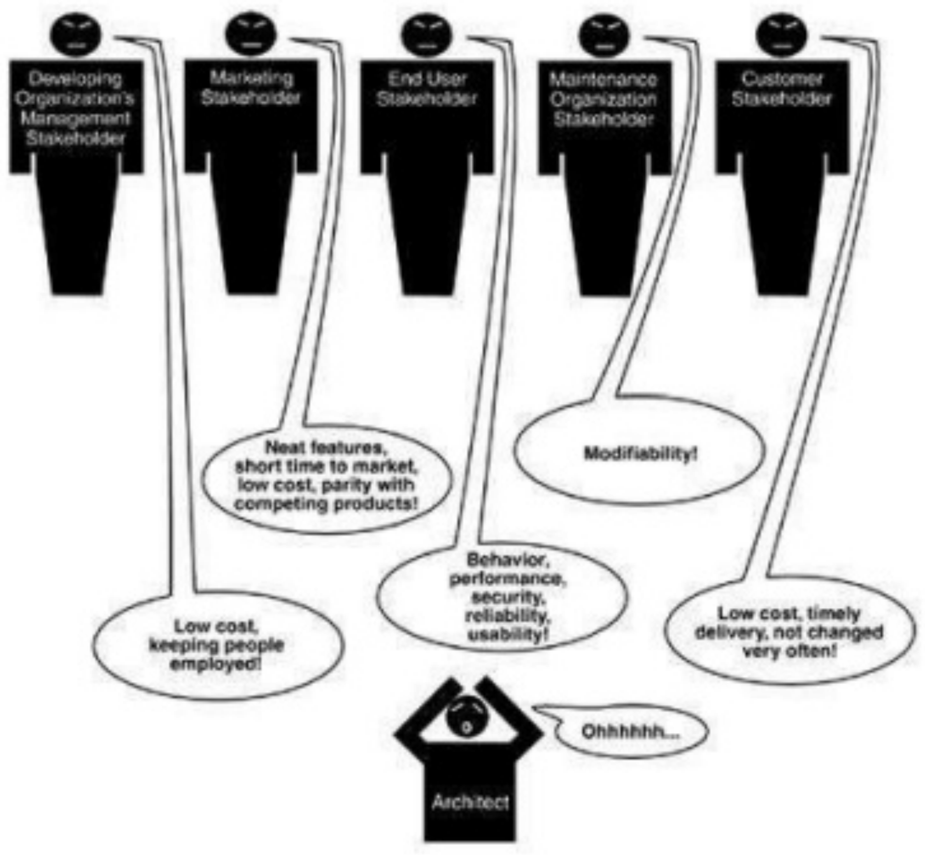
· the maintainers, and

· even those who market the system.

3. Stakeholders have different concerns that they wish the system to guarantee or optimize, including things as diverse as providing a certain behavior at runtime, performing well on a particular piece of hardware, being easy to customize, achieving short time to market or low cost of development, gainfully employing programmers who have a particular specialty, or providing a broad range of functions.

4. Figure shows the architect receiving helpful stakeholder "**suggestions.**"

. Influence of stakeholders on the architect


Developing Organization's Management Stakeholder
Marketing Stakeholder
End User Stakeholder
Maintenance Organization Stakeholder
Customer Stakeholder
Neat features, short time to market, low cost, parity with competing products!
Modifiability!
Behavior, performance, security, reliability, usability!
Low cost, keeping people employed!
Low cost, timely delivery, not changed very often!
Ohhhhhh...
Architect

5. Having an acceptable system involves properties such as performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems as well as behavior.

6. Indeed, we will see that these properties determine the overall design of the architecture.

7. All of them, and others, affect how the delivered system is viewed by its eventual recipients, and so they find a voice in one or more of the system's stakeholders.

8. The underlying problem, of course, is that each stakeholder has different concerns and goals, some of which may be contradictory.

9. Properties can be listed and discussed, of course, in an artifact such as a requirements document.

10.But it is a rare requirements document that does a good job of capturing all of a system's quality requirements in testable detail.

11. The reality is that the architect often has to fill in the blanks and mediate the conflicts.

ARCHITECTURES ARE INFLUENCED BY THE DEVELOPING ORGANIZATION:

1. In addition to the organizational goals expressed through requirements, an architecture is influenced by the structure or nature of the development organization.

2. For example, if the organization has an abundance of idle programmers skilled in client-server communications, then a client-server architecture might be the approach supported by management.

3. If not, it may well be rejected. Staff skills are one additional influence, but so are the development schedule and budget.

There are three classes of influence that come from the developing organization: **immediate business, long-term business, and organizational structure**.

· An organization may have an immediate business investment in certain assets,

such as existing architectures and the products based on them. The foundation of a development project may be that the proposed system is the next in a sequence of similar systems, and the cost estimates assume a high degree of asset re-use.

· An organization may wish to make a long-term business investment in an infrastructure to pursue strategic goals and may view the proposed system as one means of financing and extending that infrastructure.

· The organizational structure can shape the software architecture. In the case study in (Flight Simulation: A Case Study in Architecture for Integrability),

the development of some of the subsystems was subcontracted because the subcontractors provided specialized expertise. This was made possible by a division of functionality in the architecture that allowed isolation of the specialities.

ARCHITECTURES ARE INFLUENCED BY THE BACKGROUND AND EXPERIENCE OF THE ARCHITECTS:

1. If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort.

2. Conversely, if their prior experience with this approach was disastrous, the architects may be reluctant to try it again.

3. Architectural choices may also come from an architect's education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well.

4. The architects may also wish to experiment with an architectural pattern or technique learned from a book (such as this one) or a course.

ARCHITECTURES ARE INFLUENCED BY THE TECHNICAL

ENVIRONMENT:

1. A special case of the architect's background and experience is reflected by the technical environment.

3. The environment that is current when an architecture is designed will influence that architecture.

4. It might include standard industry practices or software engineering techniques prevalent in the architect's professional community.

5. It is a brave architect who, in today's environment, does not at least consider a Web-based, object-oriented, middleware-supported design for an information system.

RAMIFICATIONS OF INFLUENCES ON AN ARCHITECTURE:

1. Influences on an architecture come from a wide variety of sources. Some are only implied, while others are explicitly in conflict.

2. Almost never are the properties required by the business and organizational goals consciously understood, let alone fully articulated.

3. Indeed, even customer requirements are seldom documented completely, which means that the inevitable conflict among different stakeholders' goals has not been resolved.

4. However, architects need to know and understand the nature, source, and priority of constraints on the project as early as possible.

5. Therefore, they must identify and actively engage the stakeholders to solicit their needs and expectations.

6. Without such engagement, the stakeholders will, at some point, demand that the architects explain why each proposed architecture is unacceptable, thus delaying the project and idling workers.

7. Early engagement of stakeholders allows the architects to understand the constraints of the task, manage expectations, negotiate priorities, and make tradeoffs.

8. Architecture reviews and iterative prototyping are two means for achieving it.

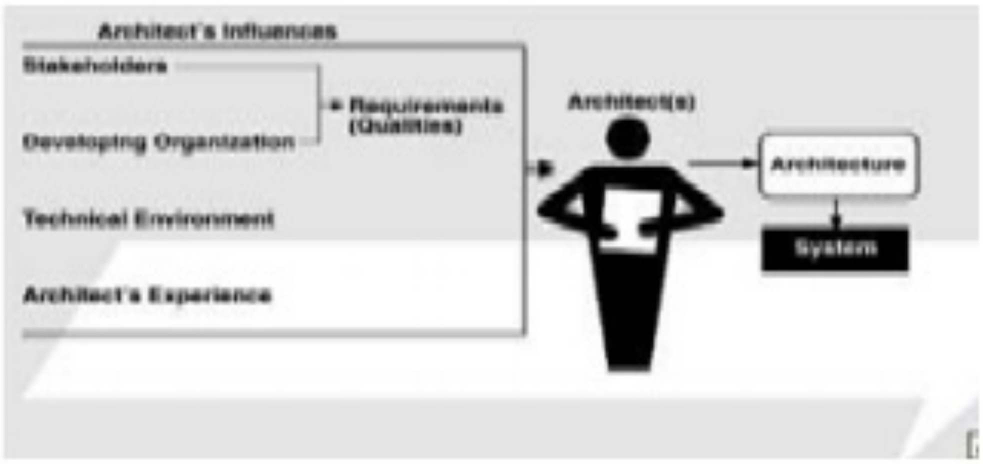
9. It should be apparent that the architects need more than just technical skills.

10.Explanations to one stakeholder or another will be required regarding the chosen priorities of different properties and why particular stakeholders are not having all of their expectations satisfied.

11. For an effective architect, then, diplomacy, negotiation, and communication skills are essential.

12 .The influences on the architect, and hence on the architecture, are shown in figure. Architects are influenced by the requirements for the product as derived from its stakeholders, the structure and goals of the developing organization, the available technical environment, and their own background and experience.

Influences on the architecture



THE ARCHITECTURES AFFECT THE FACTORS THAT INFLUENCE THEM:

1. The main message of this book is that the relationships among business goals, product requirements, architects' experience, architectures, and fielded systems form a cycle with feedback loops that a business can manage.

2. A business manages this cycle to handle growth, to expand its enterprise area, and to take advantage of previous investments in architecture and system building.

3. Some of the feedback comes from the architecture itself, and some comes from the system built from it.