Mc Graw Hill Education

Because learning changes everything.®

Chapter 1

Software and Software Engineering

Introduction

Nature of Software – Defining Software

Software is:

1) *Instructions (computer programs) that when executed provide desired features, function, and performance;*
2) *Data structures that enable the programs to adequately manipulate information.*
3) *Documentation that describes the operation and use of the programs.*

What is Software?

- *Software is developed or engineered it is not manufactured in the classical sense.*
- *Software doesn't "wear out“ but is does deteriorate.*
- *Although the industry is moving toward component-based construction, most software continues to be custom-built.*

Software Application Domains

- System software.
- Application software.
- Engineering/Scientific software.
- Embedded software.
- Product-line software.
- Web/Mobile applications.
- AI software (robotics, neural nets, game playing).

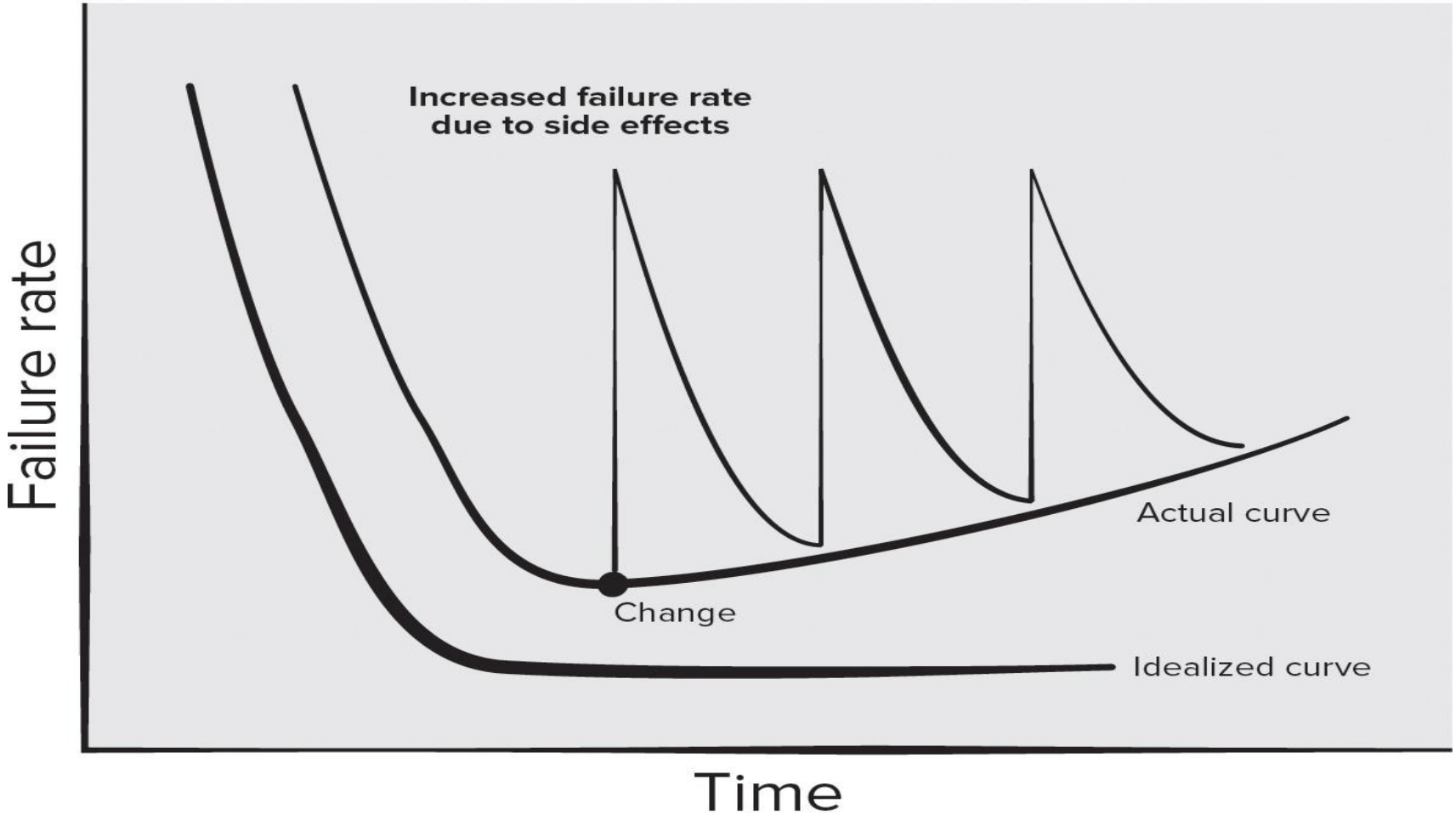
소프트웨어 위기의 등장

- 소프트웨어 크라이시스(Crisis)
  - 컴퓨터에 의한 계산 용량과 문제의 복잡도가 급증하면서, 새로운 소프트웨어 개발 방법의 필요성을 인식

- 소프트웨어 위기의 원인
  - 소프트웨어 규모의 대형화 및 복잡도 증가에 따른 개발 비용 증대
  - 소프트웨어 유지보수의 어려움과 개발 정체 현상 발생
  - 소프트웨어 개발 프로젝트 기간 및 소요 예산에 대한 정확한 예측의 어려움
  - 신기술에 대한 교육 및 훈련의 부족
  - 사용자의 소프트웨어에 대한 기대치 증가
  - 소프트웨어에 대한 사용자 요구사항의 빈번한 변경 및 반영

왜 소프트웨어 개발이 어려운가

- 의사소통의 문제
  - 소프트웨어 개발 과정에 참여하는 다양한 역할자(프로젝트 관리자, 품질 관리자, 개발자, 사용자 등)간의 의사 소통 오류
- 시스템의 순차 특성
  - 3차원적인 실세계의 서비스를 2차원 평면에서 프로그램 코드로 나열
- 개발에 의한 결과물
  - 조립이나 공식에 의한 문제 풀이가 아닌 개발자의 지적 활동에 의한 산출물
- 프로젝트 복잡성
  - 프로젝트마다 개발 기간, 개발자 수, 사용자 수준 등의 차이로 인하여 유연한 관리 필요
- 다양한 관리 이슈
  - 요구사항 변경관리, 일정 관리, 코드 버전 관리 등 다수의 활동간의 오케스트레이션



Access the text alternative for slide images.

소프트웨어 고장 그래프

- 하드웨어 고장 그래프

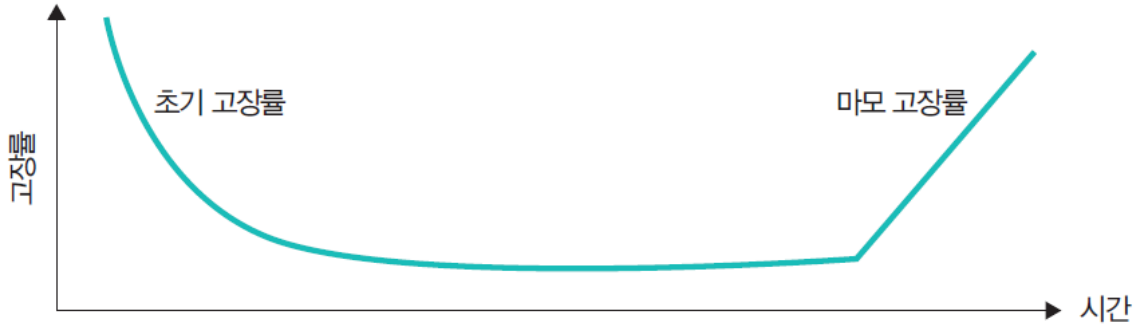


그림 1-11 하드웨어 수명주기에 따른 고장률 그래프

- 소프트웨어 고장 그래프

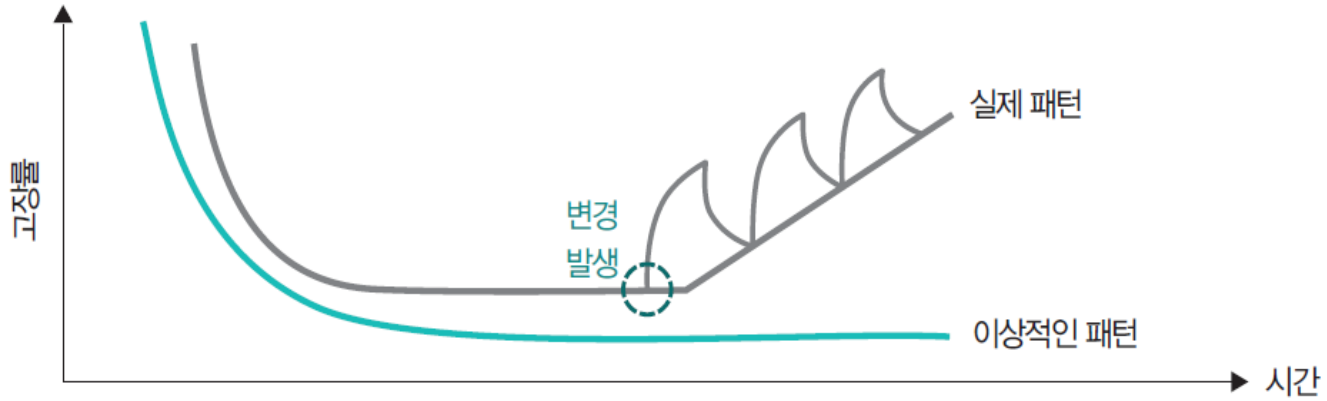


그림 1-12 소프트웨어 수명주기에 따른 고장률 그래프

Legacy Software

Why must software change?

- Software must be *adapted* to meet the needs of new computing environments or technology.
- Software must be *enhanced* to implement new business requirements.
- Software must be *extended* to make it interoperable with other more modern systems or databases.
- Software must be *re-architected* to make it viable within a network environment**.**

Defining the Discipline

The IEEE definition:

Software Engineering:

1. *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
2. *The study of approaches as in (1).*

소프트웨어 공학의 정의

■ 소프트웨어공학이란?

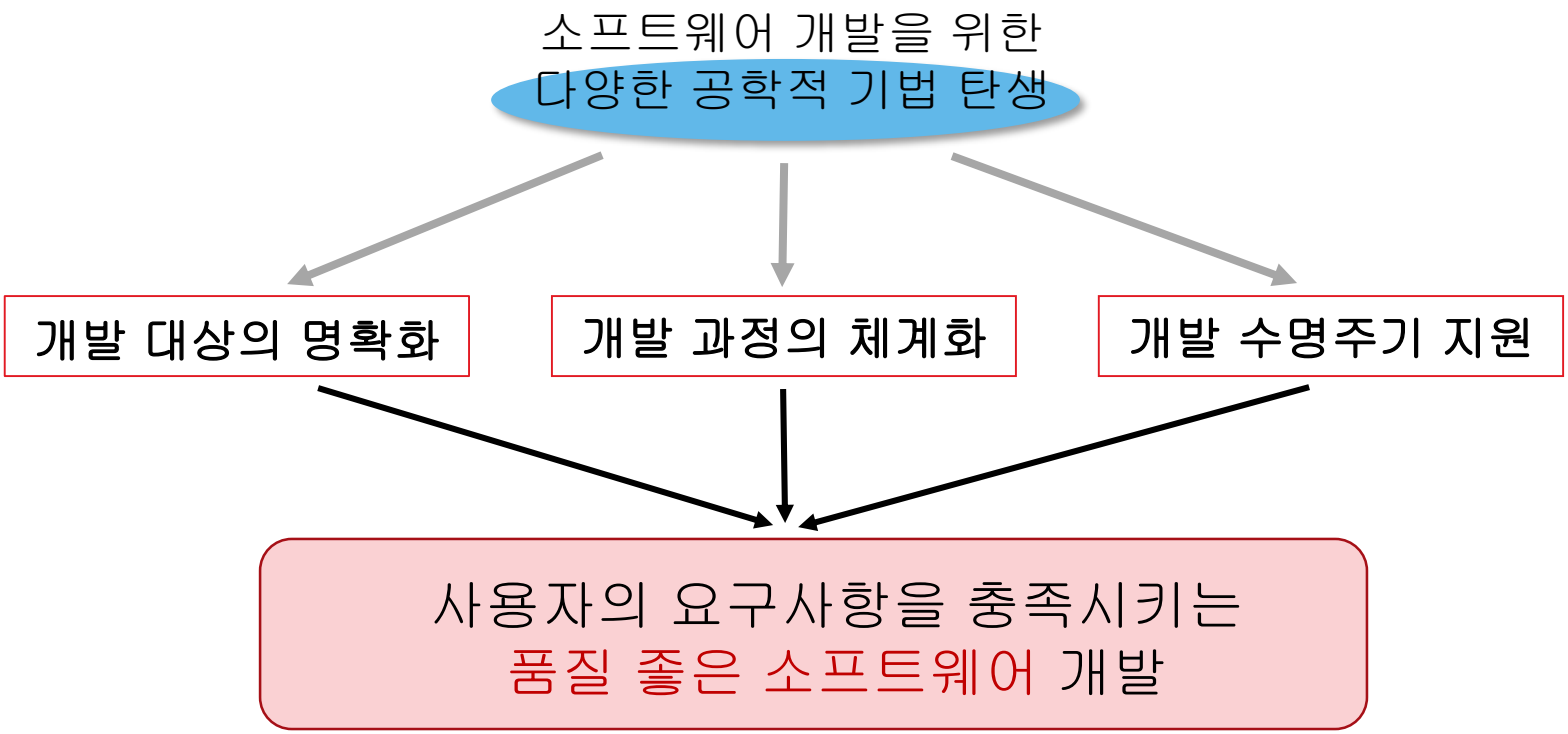
번호	제안자(연도)	소프트웨어 공학의 정의
1	Bauer (1972)	Deals with establishment of sound engineering principles and methods in order to economically obtain software that is reliable and works on real machines. (실제 환경에서 신뢰성 있는 소프트웨어를 경제적으로 확보하기 위한 건전한 공학적 원리와 방법의 구축 및 적용)
2	Boehm (1976)	Practical application of scientific knowledge in the design and construction of computer program and the associated documentation required to develop, operate, and maintain them. (컴퓨터 프로그램을 개발, 운영, 유지보수하기 위해 요구되는 프로그램의 설계 및 구축과 관련된 문서화에 대한 과학적 지식의 실질적인 적용)
3	Zelkowitz (1978)	Study on the principles and methodologies for developing and maintaining software systems. (소프트웨어 시스템을 개발하고 유지보수하기 위한 원리 및 방법에 대한 연구)

소프트웨어 공학의 정의

■ 소프트웨어공학이란?

번호	제안자(연도)	소프트웨어 공학의 정의
4	Parnas (1978)	Multi-person construction of multi-version software. (소프트웨어의 여러 버전에 대한 여러 사람에 의한 개발)
5	Pfleeger (1990)	Methods and techniques to develop and maintain quality software to solve problem. (주어진 문제를 해결하는 품질 좋은 소프트웨어를 개발, 유지보수하기 위하여 적용되는 방법 및 기법)
6	ISO 24765 (2010)	Systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software. (소프트웨어의 설계, 구현, 테스트, 문서화를 위한 과학적이고 기술적인 지식, 방법, 경험의 체계적인 적용)

소프트웨어 공학의 목표



소프트웨어 공학의 원리

- 엄격성과 정형성
  - 소프트웨어는 개발자의 경험과 지식에 의존적인 창의적, 공학적 활동의 산출물
  - 소프트웨어 개발은 주어진 시간과 비용에서 명확하게 개발되어어야 하는 엄격성

- 관심사의 분할
  - 복잡한 문제를 단순한 문제로 분리하여 적용하는 소프트웨어 개발 활동
  - 소프트웨어 개발 과정의 분할 : 요구사항 분석 > 설계 > 구현 > 테스팅 등의 단계로 분할
  - 소프트웨어 테스트 과정의 분할 : 단위 테스트 > 통합 테스트 > 시스템 테스트 등으로 분할

- 모듈화
  - 독립적인 기능을 갖는 프로그램의 조작
  - 높은 응집력과 낮은 결합력을 갖는 소프트웨어 구조 설계
  - 이해도를 높이고 변경 영향을 최소화하기 위한 공학적 원리

소프트웨어 공학의 원리

- 추상화
  - 세부사항은 감추고 대표적인 속성으로 대상물을 정의하는 공학적 원리
  - 대상물에 대한 직관적인 이해를 높이고 관심사를 잘 표현할 수 있음.
  - 예: 함수 정의, 매크로 함수, 객체, 추상데이타 타입 등

- 변경의 예측
  - 소프트웨어 개발과정에서 변경 발생은 피할 수 없는 사건
  - 변경을 대처하기 위한 공학적 방법의 적용이 요구됨.
  - 변경이 발생할 것으로 예상되는 부분을 모듈화 구조로 분리

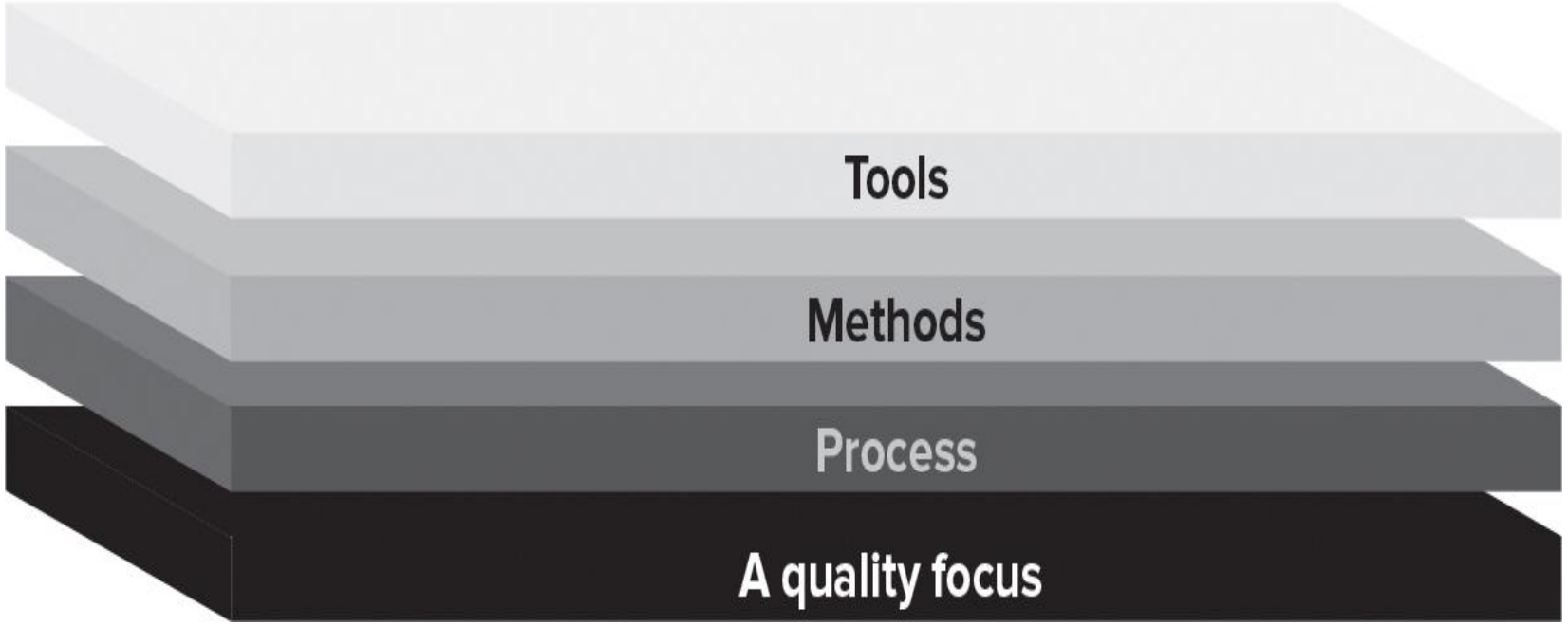
- 일반화
  - 다양한 플랫폼, 다양한 환경, 다양한 사용자를 지원하기 위한 원리
  - 하드웨어 성능이나 사양에 의존적인지 않는 소프트웨어 개발

소프트웨어 공학의 원리

- 점진성
  - 단계적이며 순차적으로 소프트웨어를 개발하고자 하는 공학적 원리
  - 작은 단위의 소프트웨어를 반복 개발하면서 전체 시스템을 완성
  - 단계적인 개발과 배포를 통한 사용자 피드백의 수집과 반영

- 명세화
  - 소프트웨어 개발 과정 및 대상물에 대한 정보를 체계적으로 기술하는 원리
  - 팀 기반의 개발 활동을 지원하기 위한 정보의 공유 지원
  - 지속적으로 진화하는 소프트웨어에 대한 이력서

Software Engineering Layers



Process Framework Activities

Communication.

Planning.

Modeling.

- Analysis of requirements.
- Design.

Construction:

- Code generation.
- Testing.

Deployment.

Umbrella Activities

- Software project tracking and control.
- Risk management.
- Software quality assurance.
- Technical reviews.
- Measurement.
- Software configuration management.
- Reusability management.
- Work product preparation and production.

Essence of Software Engineering Practice

Polya suggests:

1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine result for accuracy* (testing & quality assurance).

Understand the Problem

- *Who has a stake in the solution to the problem?*

  That is, who are the stakeholders?

- *What are the unknowns?*

  What data, functions, and features are required to properly solve the problem?

- *Can the problem be compartmentalized?*

  Is it possible to represent smaller problems that may be easier to understand?

- *Can the problem be represented graphically?*

  Can an analysis model be created?

Plan a Solution

- *Have you seen similar problems before?*

  Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?

- *Has a similar problem been solved?*

  If so, are elements of the solution reusable?

- *Can subproblems be defined?*

  If so, are solutions readily apparent for the subproblems?

- *Can you represent a solution in a manner that leads to effective implementation?*

  Can a design model be created?

Carryout the Plan

- *Does the solution conform to the plan?*

  Is source code traceable to the design model?

- *Is each component part of the solution provably correct?*

  Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

Examine the Result

- *Is it possible to test each component part of the solution?*

  Has a reasonable testing strategy been implemented?

- *Does the solution produce results, that conform to the data, functions, and features that are required?*

  Has the software been validated against all stakeholder requirements?

David Hooker's General Principles

1. *The Reason It All Exists* – provide value to users.
2. *KISS (Keep It Simple, Stupid!)* – design simple as it can be.
3. *Maintain the Vision* – clear vision is essential.
4. *What You Produce, Others Will Consume.*
5. *Be Open to the Future* - do not design yourself into a corner.
6. *Plan Ahead for Reuse* – reduces cost and increases value.
7. *Think!* – placing thought before action produce results.

How it all Starts

Every software project is precipitated by some business need—

- The need to correct a defect in an existing application;
- The need to the need to adapt a ‘legacy system’ to a changing business environment;
- The need to extend the functions and features of an existing application;
- The need to create a new product, service, or system.