**소프트웨어 공학**: 소프트웨어의 품질과 생산성을 향상시키기 위하여 사용자의 요구사항을 체계적으로 분석하여 설계, 구현, 테스트, 유지보수, 폐기까지의 소프트웨어 전 수명주기간에 걸쳐 이루어지는 체계적인 접근법(컴퓨터 과학 지식을 기반으로 사회적, 경제적 맥락에서 소프트웨어를 만들어내는 것)

**Program**: 실행가능한 컴퓨터 명령어의 집합

**Software**: 컴퓨터 프로그램 + 관련된 문서들

**소프트웨어 위기**: 소프트웨어의 사이즈, 복잡성이 증가되었음에도 불구하고 규칙없이 소프트웨어를 개발하여 발생, Late Delivery, Over budget, inconsistent with the specification, Difficult to understand, Too Costly to modify or improve

**소프트웨어 공학의 goal**: 사용자의 요구사항을 만족하는 퀄리티 있는 소프트웨어를 시간, 예산 안에 생산하고 이를 통해 비즈니스 이익을 만들어내는 것

**소프트웨어 공학의 3요소**

* People: 소프트웨어 프로젝트에 참여하는 사람에게는 각자의 책임과 역할이 있다.
* Technology: 소프트웨어 프로젝트의 전 수명주기에 걸쳐 적용되는 방법 및 기법
* Process: 다양한 프로젝트 환경 및 여건하에서 체계적으로 소프트웨어를 개발할 수 있도록 가이드라인을 제공

**소프트웨어 공학 기술 범위**

* 요구사항, 설계, 구현, 시험, 유지보수, 형상관리, 공학 관리, 공학프로세스, 공학 도구 및 방법, 품질

**개발 기술**: 요구공학, 아키텍처&디자인, 테스트, 유지보수, 정형기법,재사용

**지원 기술**: 프로젝트 관리, 프로세스, 도구 및 환경

**Software Lifecycle Model**: 소프트웨어 프로젝트의 시작부터 종료까지 단계 단계를 정의하고 있는 것

* people, overall process, intermediate products, stages of process
* 요구사항 -> 분석 -> 설계 -> 구현 -> 테스팅 -> 유지보수

**Hacking**: 분석, 설계, 목표 없이 구현에 집중하는 것

**Waterfall**: 이전 단계를 완전히 완료해야만 다음 단계로 진행하는 모델

* 관리자가 프로젝트 단계별로 진척사항을 잘 control 할 수 있음
* 사용자는 처음에 요구사항을 완벽하게 다 표현해야함
* 개발자는 프로젝트 도메인에 있어서 전문가여야 함
* 설계 전에 요구사항이 완료되어야 하며, 코딩 전에 설계가 완료되어야 한다.
* 마지막의 통합과 테스팅에 굉장히 많은 노력, 비용이 들어간다.(마지막에 한번에 해서)
* system이 끝나기 전에는 software capability를 파악할 수 없다.
* Requirements Definition-> software design-> implementation and unit testing-> Integration and system testing-> operation and maintenance

**Incremental**: System을 우선순위에 따라 여러 기능으로 나누어 우선 순위에 따라 기능을 시스템에 붙여나가는 모델

* 가장 중요한 기능일 수록 먼저 개발됨
* 진행 중에 계속해서 demonstration을 할 수 있음.
* 처음에 붙여진 기능이 많이 테스트 됨
* 이후 버전에 일부 변경사항을 통합할 수 있음
* incremental delivery를 가능하도록 함
* 사용자로부터 일찍 피드백을 받을 수 있음

**Evolutionary**: 요구사항을 사전에 명세할 수 없을 때 분석, 설계, 구현, 테스트, 평가의 과정을 한 사이클로해서 반복하는 것

* 프로젝트가 실현불가능해 취소된 경우, 제품이 완성되어 전달된 경우, 요구사항을 알게되어 incremental로 바뀐 경우 cycle이 종료됨
* High Risk Project 의미

**Spiral**: 프로토타입을 만들어 낼때마다 Risk 분석을 하는 모델

* risk 평가에 대한 개념을 소개(장점)
* risk management를 소프트웨어 개발 프로세스에 도입(장점)
* 크고 무거운 iteration을 수행하여야 하며, 매 iteration마다 product를 평가해야함(단점)
* 소프트웨어 프로젝트가 복잡해짐(단점)

**Unified Software Development Process**

* UML사용
* use-case driven, architecture centric, incremental and iterative
* Phase: Inception, Elaboration, Construction, Transition(시간 단위로 구성)
* Workflow: Requirement, Analysis, Design, Implementation, Test(태스크 단위로 구성)
* 각 phase마다 iteration들 있음.

**Extreme Programming**

* Agile: 작은 team을 위한 lightweight process 진행
* change가 빠르고 쌈, overhead가 낮음, 요구사항 변화에 강함, system을 간단하게 함
* Customer가 피드백을 자주해줘야 함, 복잡한 프로젝트에는 맞지 않음, 자동 테스트 되지 않는 프로그램에는 적합하지 않음
* 작은 반복을 여러 번 수행함
* incremental model 기반
* cost benefit 분석에 의해 initial features 결정
* Programming Pair를 통해 구현
* 설계를 강조하지 않음
* Continuous integration, testing을 많이 수행
* Collective ownership, open workspace

**Object:** 어떤 사물이나 개념, 소프트웨어 entity와 같이 다른 것과 구분되어지는 독립된 하나의 entity로 state와 behavior 가지고 있음

* 객체는 unique한 identity를 가지고 있음
* state는 객체가 존재할 수 있는 가능한 condition
* behavior는 object가 어떻게 행동할지를 결정

**Object Orientation**

* Abstraction: 본질적인 특징을 단순화 시켜서 나타내는 것
* Modularity: 복잡한 것을 관리 가능한 작은 조각으로 쪼개는 것
* Encapsulation: 구현을 Client로부터 숨기는 것
* Hierarchy: 부모 자식간의 상속 관계

**Class**: 같은 attribute, relationship, operation, semantic을 공유하는 객체들의 집합의 description

* object는 Class의 instance
* Class는 object의 abstract definition
* attribute: 데이터 의미
* operation: 기능 의미

**Polymorphism**: 서로 다른 implementation을 하나의 interface안에 숨기는 것(하나의 interface를 통해 서로 다른 implementation들을 사용할 수 있도록 하는 것)

**Interface** : Polymorphism과 plug-and-play를 가능하게 함.

Association: 하나이상의 객체가 서로 구조적인 관계가 있음을 의미

Multiplicity: 한 클래스 안의 하나의 객체가, 다른 클래스의 객체와 관련 맺고 있는 숫자

Aggregation: 하나 이상의 객체 사이에서 나타나는 전체와 부분의 관계

Composition: 하나이상의 객체 사이에서 나타나는 전체와 부분의 관계, lifetime이 밀접한 관련

Dependency: 하나이상의 객체 사이에서 나타나는 일시적인 관계, 구조적인 관계는 아님

Generalization: 상속의 관계

**UML**

* OO requirement, analysis, design의 product를 묘사하는 graphical language
* Object Oriented Development의 표준

**COMMET :** UML을 가지고 어떻게 설계할 것인지에 대한 방법론

* concurrent, distributed, real-time application 설계를 위한 기준
* Requirement Modeling-> Analysis Modeling-> Design Modeling-> Incremental Software Construction-> Incremental Software Integration->System Testing
* + Incremental Prototyping, Throwaway Prototyping

**Requirement Modeling**

* Use case Modeling: use case와 actor 측면에서 Software의 기능적인 요구사항을 정의
* Use case Diagram이 결과물로 나오게 됨

**Analysis Modeling** : 요구사항의 구체적인 내용을 ‘이해’하는 것

* Static Modeling
* Class들 사이의 구조적 관계를 정의
* Class와 그들의 관계를 Class Diagram으로 표현
* Dynamic Modeling
* object들이 use case에 어떻게 participate하는지 collaboration diagram과 sequence diagram을 통해 정의

**Design Modeling**: 요구사항에 대한 Solution을 만들어내는 것

* 종합적인 Software Architecture 개발
* System을 Subsystem으로 구조화

**Incremental Development**

* Incremental Software Construction
* Use case기반으로 system의 subset을 골라냄
* Incremental Software Integration
* increment마다 통합 테스트
* 각 use case마다 test case 개발
* whitebox testing(use case안의 object사이의 interface test)

**System Testing**

* 시스템의 기능적인 testing
* Black box testing(use case 기반)
* 독립적인 test team

**Use Case Modeling**: use case와 actor의 측면에서 시스템의 기능적인 요구사항을 정의

* **Use case**: 사용자와 시스템 사이에서 일어나는 interaction들의 sequence
* use case는 design model에서 정제되며 system test case와 integration의 기반
* 서로 loosly coupled, 내부는 strongly cohesive, 크기는 equally distributed
* **Actor:** system과 직접적으로 상호작용하는 external entity
* Human user, External I/O device, External System, Timer
* Actor는 I/O device나 external system을 이용해 시스템과 상호작용
* Actor가 use case를 initiate 함
* Primary Actor: system에 input을 제공하여 use case start
* Secondary Actor: use case에 참여, 다른 use case의 primary actor가 될 수 있음
* 같은 방식으로 system을 사용하는 모든 user들을 대표

**Documenting Use cases**

* Name, Summary, Dependency, Actors, Preconditions, Description, Alternatives, Postcondition

**Use Guidelines**

* System을 Black box라고 생각
* 능동적으로 표현
* use case는 interaction들의 sequence이지 작은 function이 아님
* if-else, loop를 use case description에서 사용하지 않음

**Relationships**

* Include Relationship: 여러 use case에서 공통으로 나타나는 패턴을 abstract use case로 추출하여 concrete use case가 abstract use case를 include함
* Extend Relationship: 특정 상황에서 어떤 use case는 다른 use case의 description으로 확장됨

**Use Case Package**

* 여러 관련된 use case들을 grouping

기타.

**Requirement engineering**이 중요한 이유

* 대부분의 error는 초기단계에서 발생
* 요구사항은 시스템 개발에 많은 영향을 끼침

**Software Essential Problem**

* Complexity: 소프트웨어 개체들은 그 크기에 비해 인간이 만든 어떤 구조물 보다 복잡
* Conformity: 수많은 사람들이 인터페이스를 각자 다르게 설계하여 여러 인터페이스에 맞추기 힘듦
* Changeability: 소프트웨어는 지속적으로 변화하는 것들에 둘러싸여 있음
* Invisibility: 소프트웨어는 눈에 보이지 않으며 구조를 시각화하기 어려움.

**USDP 보충**

* **use case driven**: usecase가 기능적 요구사항을 나타내며 Iteration의 내용을 정의한다.
* **architecture centric**: 시스템을 구축하기 위한 작업의 중심에 architecture가 있음
* **iterative and incremental**: elaboration, construction, transition은 여러 번의 iteration으로 이루어져 있음
* **inception:** 프로젝트에 대한 사업성을 판단하고, 범위를 설정하여 전체적인 계획 및 비용을 산정한다.
* **elaboration**: 대부분의 요구사항이 정리되고, 중요한 구성요소들을 구현한다.
* c**onstruction**: 모델과 시스템을 구축한다.
* **transition**: 납품, 설치, 운영테스트를 통한 성능 조정 작업과 사용자 교육등을 수행한다.