Orientation(해외 사례)

실제 여러 소프트웨어적으로 발생한 문제들 소개
- 추후에 나오는 여러 소프트웨어 개념을 적용시켜 설명할 수 있어야 함.-

1. Youtube 서비스 중단 - 갑자기 유튜브 서비스 먹통
- 실시간, 생중계 콘텐츠 손해 - 각종 문의 묵묵부답.
스토리지 자원할당 시스템을 새롭게 도입하는 과정에 문제가 발생
사용자의 서비스 사용량을 0으로 입력

2. Google 서비스 중단 - 여러 가지 피해발생 그러나 법적 책임 없어 보상 어려움.
잘못된 설정값

3, NASA Challenger 폭발(중요)
문제가 발생하기 전에 이미 해당 문제가 발생할 것임을 예측한 직원이 있었음.
하지만 여러 상황적인 요소 때문에 (촉박한 데드라인, 세간의 관심 등) 으로 인해 해당 직원의 목소리는 무시당하고 강행함.

기계 결함? 물론 이게 원초적인 문제가 맞긴하지만 결국 주요 원인은 조직의 문제이다.
소프트웨어는 기본적으로 사람의 실수로 인해 문제가 발생할 가능성이 농후함. 이 문제를 해결하는 것이 중요!

4. Tesla Autopilot 오류
특정 색깔의 차량 인식 못 함. 소프트웨어적 결함.

Orientation(국내 사례)

실제 여러 소프트웨어적으로 발생한 문제들 소개
- 추후에 나오는 여러 소프트웨어 개념을 적용시켜 설명할 수 있어야 함.-
- 내가 해당 소프트웨어 관리자였다면 어떻게 해결할지 생각해볼 것 -

1. 카카오톡 서비스 장애
몇분간 카카오톡 메시지 전송이 안 됨.

2. 삼풍 백화점 붕괴(중요)
이는 예견된 문제였음. 이미 전부터 많은 관계자들이 이를 경고함.
1. 붕괴 전 진동느낌
2. 건물 벽에 균열
3. 바닥이 내려 앉음.
-> 실제로 무너질거라 생각을 못 함.
검사관이 즉시 대피하라고 했지만 이를 무시함.

3. KT 아현 지사 사고 원인 분석
서비스가 실제로 돌아가기 하지만 실제 안을 자세히 살펴보면 많은 문제가 존재함.
실제 안에 시설들이 전혀 관리되지 않고 정리 되지 않았으며 그저 방치됨.
이미 해당 시설을 증설, 보수하는 관리자는 문제가 발생할 것임을 경고
그저 하나씩 방치된 문제가 나중에 처리하기 힘들어짐.
아무리 사소한 문제라도 나중에 소프트웨어적으로 큰 문제를 야기할 수 있으므로 귀찮다고 미루지 말고 사소한 문제를 해결하도록 노력해야 함. 유지보수가 편하게!

4. 전남 영광 한빛 원전 1호기 사고 원인
제대로 교육이 되지 않은 직원이 투입됨.

Chap 1-Introduction

1. Professional software development - 소프트웨어 개발이란?
2. Software engineering ethics - 소프트웨어 윤리
3. Case studies - 예제 분석

현재 경제와 관련된 모든 산업들은 소프트웨어와 연관됨.(키오스트, 사무업무 - AI)

-Software costs-

1. 구동하기 위한 환경 + 소프트웨어 가격 (초기 비용)
2. 유지보수 비용 (이게 처음에 작아보여도 나중에 훨씬 크고 중요함!)

하드웨어적인 비용보다 소프트웨어적인 비용이 크다.

-왜 프로젝트가 실패하는가?-

오늘날 소프트웨어는 매우 복잡하고 계속해서 새로운 변화가 요구됨.
따라서 소프트웨어를 지속적으로 발전시킬 수 있도록 유지보수가 쉽게 개발되야 함.
즉, 유연성, 확장성이 매우 중요한데 처음에 이를 고려하기 힘듦.
그래서 계속 코드가 덕지덕지 추가되어 나중엔 유지보수가 거의 불가능해지는 스파게티식 코드로 프로젝트가 완성되어 유지보수가 안 되어 프로젝트 실패함.

1. Professional software development

좋은 소프트웨어란?
-> 요구 사항이 충족되고, 성능이 좋으며, 안정성이 좋고, 유지보수하기 쉬운 소프트웨어

software engineering이란?
-> 일종의 규칙이다. 소프트웨어 개발 및 생산 전반에 걸친 규칙! 룰!

가장 근본적인 software engineering 활동은?
-> specification(명세서) - development(개발) - validation(타당성 검증) - evolution(발전)

system engineering은 좀 더 포괄적인 개념으로 하드웨어 + 소프트웨어 + 인프라 등 engineering 전반에 걸친 이야기임.

●소프트웨어 엔지니어링의 key challenges는?
증가하는 복잡성, 다양한 요구를 최소한의 시간으로 대처할 수 있어야 함.

●software engineering의 cost는?
대게 60% 개발비용 - 40% 유지보수 및 발전 비용
그러나 요즘날은 후자가 훨씬 큰 경우가 많음.

Generic Product - 개발자가 명세서 정의
-원하는 사람이 있으면 이 소프트웨어를 사가라.
ex) 그래픽 디자인 툴, 프로젝트 Management 프로그램, CAD, - 고객이 해당 프로그램이 필요하면 사가는 형태

Customized Product - 특정 고객이 명세서 정의/ 요구에 맞춰서 명세서 정의
출시 전부터 특정 고객에 맞춰서 제작.
ex) 관제탑 프로그램, 교통 체증 감시 프로그램, 핵융합 발전소 프로그램 등

●소프트웨어 engineering 중요성
우리는 신뢰성이 높고 빠른 소프트웨어를 개발해야 함.
유지보수하는 비용이 매우 큼. 이를 최대한 줄이도록 노력해야 함. (코드 재사용 등)

software 개발하면서 만나게 될 일반적인 문제들

1. 매체의 다양성
점점 더 다양한 타입의 컴퓨터/ 모바일 기기를 지원할 수 있도록 개발해야 함.

2. 빠른 변화성(사회가 빠르게 변화함)
사업과 사회는 빠르게 변화하고 있음. 이에 맞춰 소프트웨어도 빠르게 변화시킬 수 있도록 개발해야 함.

3. 안전성과 보안
요즘날 개인정보가 중요해지고 민감해지면서 이에 관련된 데이터 보안/안정성이 매우 중요해짐.

4. Scale(규모 변화)
빠르게 늘어나는 요구에 맞춰 소프트웨어 Scaling이 가능해야 함.

--- 중요 ----
수많은 소프트웨어 타입이 존재하고 모든 곳에 적용할 수 있는 기술은 없음.
각 문제 상황에 맞춰 적절한 툴과 기법을 적용해야 함.

Application Types

1.Stand-alone applications - local 컴퓨터 안에서 동작하고 끝나는 소프트웨어
네트워크 연결 필요 없이 그저 내 컴퓨터 안에서 잘 작동하면 됨.

2. Interactive transaction-based applications
- remote computer에서 실행되는 application(WEB , Internet 이용)
ex) e-commerse app

3. Embedded control systems
하드웨어 장치들을 조작하고 관리하는 소프트웨어
realtime과 연관된 경우가 많음.
심박수 측정, 인슐린 투입 장치 등

4. Batch processing systems (차례대로 순차적으로 일괄처리 - 처리 대용량 처리에 자주 쓰임)
금융기관 - 밤에 기록들 한번에 처리
수능 - 시험이 끝나고 모아서 한번에 처리

5. Entertainment system
- 유저를 즐겁게 하기 위해 개발된 시스템

6. System for modeling and simulation
- 시뮬레이션을 위한 개발된 소프트웨어

7. Systems of systems
시스템의 시스템 => 회사 안에는 인사, 재무, 회계, 생산, 자원 등 관련된 소프트웨어가 수많이 존재 -> 이를 한단계 위에서 총괄하는 소프트웨어

Software engineering fundamentals - 모든 곳에 쓰일 수 있는 원칙
-한 프로세스에는 하나의 작업만!
-기능성과 의존성은 매우 중요
-명세서와 요구사항 이해는 매우 중요
-소프트웨어 재사용성 증대는 매우매우 중요! (새롭게 만들기보단 재사용하자!)

Cloud computing이란 컴퓨터 서비스를 remote하게 이용하는 것.
local computer에서 작동하는 service가 아닌 remote server computer에서 작동함.

중요한 Software engineering(WEB) - 내 생각엔 WEB뿐만 아니라 다른 software가능

★software reuse

소프트웨어 재사용은 매우 중요한 접근법이다.
소프트웨어를 개발할 때 어떻게 이미 존재하는 코드를 재사용할지 이를 어떻게 취합할지 고민해야 함!!

●Incremental and agile development
중요한 핵심부분을 짜고 시장의 반응을 보면서 부가적인 것을 계속 추가하는 방식
빠른 피드백이 가능함!
점진적인 개발 방법 + 애자일 개발 방법은 매우 좋다!
시장의 반응을 빠르게 피드백하여 적용할 수 있고, 추가적인 확장이 용이하다.

●Service-oriented systems
옛날에는 거대한 하나의 시스템을 만듦.
하지만 현재는 잘게 쪼개서 여러 개의 component 개발하는 식으로 만듦.
수많은 기능을 제공하는 다양한 서비스 component를 만들고 이를 하나로 묶어서 서비스를 제공!

2. Software engineering ethics(소프트웨어 윤리의식)

소프트웨어 개발할 때는 기술만이 아닌 윤리의식을 가지고 있어야 함!!!
소프트웨어 개발자는 책임감을 가지고 정직하게 개발하며 윤리의식을 지켜야 함!

1. Confientiality(기밀성)
- 엔지니어는 기밀을 유지해야 한다! 엔지니어는 고객들의 민감한 정보를 얻을 수 있기 때문에 이 기밀유지를 잘해야 한다. 뿐만 아니라 회사 소프트웨어를 개발하면서 알게 된 여러 회사 기밀을 지켜야 한다!!

2. Competence - 자신이 할 수 없는데 할 수 있다고 하면 안 됨!
3. Intellectual property rights - 지적재산권을 잘 지켜야 한다!
4. Computer misuse - 회사 장비를 정해진 용도 외에 사용하면 안됨!!
회사의 그래픽 카드로 비트코인 채굴....

ACM/IEEE Code of Ethics - 개발자의 윤리의식을 정리한 기관
http://sigsoft.or.kr/%EC%86%8C%EC%82%AC%EC%9D%B4%EC%96%B4%ED%8B%B0-%EC%86%8C%EA%B0%9C/%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4%EA%B3%B5%ED%95%99-%EC%9C%A4%EB%A6%AC%EA%B0%95%EB%A0%B9/
이 사이트 참고

윤리의식 딜레마
- 정책 및 시니어 개발자에 의해서 원칙 못 지키는 경우
- 당신의 직원이 비윤리적인 방식으로 소프트웨어 개발하고 배포함.
- 군대 무기 시스템이나 핵 시스템 개발에 참여하는 경우

3. Case studies(사례 분석)

- 이후 본 강의 진행하면서 나오게 되는 여러 사례들 소개 -

1. insulin 주입 프로그램 - embedd system - 매우 정확하게 작동해야 함!
- 모든 센서들 정보가 정확하게 수집되어야 하며 정확한 양의 인슐린이 투입되도록 신뢰성이 높아야 함!!!
2. mental health case - 사람들의 정신건강 관리 시스템
- 헬스 매니저들에게 환자에 대한 정보가 주어져야 함. (좋은 멘탈케어를 위해)
- 스태프들에게 제때에 필요한 정보를 제공해야 함.
- 개인 맞춤형 서비스
- 환자 감시 시스템
- Privacy & Safety가 중요함!!!(개인정보가 많이 다뤄짐. 안전 매우 중요!!)
3. wilderness weather station - 기후 변화 정보 모으는 시스템
정부는 여러 황무지에 weather station을 배치하고 이 정보를 모아야 함.
각 station은 여러 측정 기구들을 통해 기온, 기압, 날씨 등 다양한 정보를 수집함.
4. iLearn - 교육 프로그램
교육에 특화된 프로그램 - learner들에게 필요한 교육을 제공!
teacher들이 원하는 교육 환경을 선택할 수 있음.
예를 들어 숙제 제출 확인을 위한 spreadsheet
이 시스템은 service-oriented system임
모든 시스템 컴포넌트는 교체될 수 있음.
점진적으로 새로운 서비스가 추가될 수 있도록 개발되야 함.
빠르게 설정을 바꿀 수 있음. 예를 들어 특정 컨텐츠를 senior 학생들만 볼 수 있게 설정.

4. Key points

1. software engineering은 소프트웨 생산 모든 측면에 관련된 규칙, 원칙이다.
2. 소프트웨어는 필수적으로 유지보수성, 의존성, 보안성, 효율성, 접근성을 충족시켜야 한다.
3. 수많은 타입의 시스템이 존재하고 각각 적절한 방법이 존재하다. 개발자가 이에 맞춰 적절한 개발 방법 및 도구들을 사용해야 한다.
4. 하지만 근본적인 소프트웨어 아이디어는 모든 곳에 적용이 가능하다.

ex) 재사용성, 요구 사항 이해, 유지보수 가능하게

5. 개발자는 윤리의식을 지키고 책임감을 가져야 한다!

Chap 2 - Software Processes

1. Software process models
2. Process activities
3. Coping with change
4. Process improvement

Plan-driven process - 계획/일정에 맞추어 개발하는 방식
ex) 공학에서 자주 사용되는 방식(공정이 끝나야 다음 단계 가능)
원자력 발전소 프로그램 (모든 검증이 끝나야 개발 가능)

각 이전 단계가 완료되어야 다음 단계 진행 가능 (문서화가 매우 중요한 개발방식)
Agile process - 변화를 빠르게 반영하며 개발하는 방식 계획보단 개발에 초점을 맞춘 방식 제때 주요한 기능만 작동하면 됨. 모든 기능이 작동할 필요 없음. 일단 당장 출시하고 사람들의 반응에 맞추어 점진적인 개발 진행

뭐가 더 좋은 개발방법이다. 이런 건 없음. 상황에 맞추어 사용할 것!!

1. Software process models

Plan-driven & Agile

1. WaterFall Model
- Plan-driven model -> 위에서 쏟아지는 폭포 생각(순차적으로 진행)
- 모든 개발 단계가 문서화되며 구분되어 진행됨. 이전 단계가 완료되어야 다음 단계 가능.
- 만약 중간에 수정이 필요하면 처음으로 돌아가서 다시 개발을 시작해야 함. 변화 반영힘듦.
- Embed System , critical system, large system에 자주 사용!

새로운 변화를 적용하는 것보다 안정성이 요구되는 시스템에 자주 사용되는 개발 모델임!!

2. Incremental development
- 점진적으로 계속해서 소프트웨어 개발하는 방식. (Plan driven, agile 둘 다 가능)
- 만약 수정 사항이 들어오면 해당 단계만 수정하면 됨!
- 이 개발 방법은 고객의 피드백 수용이 좀 더 수월함.

문제
- WaterFall과 달리 문서화가 되지 않아 프로세스 관리가 힘들다. 그리고 실제 이 프로젝트를 관리하는 입장에서 진행 단계를 확인할 수 없어 단계적 관리 힘듦.
- 소프트웨어가 점점 추가되는 방식이다 보니 스파게티 코드가 발생할 수 있으며 확장성이 떨어질 수 있다.
- 또한 개발 속도가 빠르다 보니 놓치는 게 생길 수 있다. (회사 내부 규정, 여러 오픈소스 라이센스 문제 등등)

3. Integration and cofiguration
이미 존재하는 여러 서비스, 컴포넌트들을 연결시키며 개발하는 방식
재사용에 중점을 둔 개발방식 - 이미 존재하는 소프트웨어, 애플리케이션을 가져와서 사용
COTS - Commercial-off-the-shelf 개념 등장!
기존에 소프트웨어들을 짜깁기 하는 느낌의 개발 방법!
재사용은 현재 많은 시스템의 기본적인 접근법이 됨!
ex) 라이브러리 재사용, Web API, serverless, cloud system 등
다양한 재사용 예시 존재!
이점과 단점
- 이미 많은 사람들로부터 검증이 되어 위험과 비용이 낮다.
- 재사용을 하기 때문에 빠른 개발이 가능하며 빠른 시간에 배포 가능
- 소프트웨어 대한 통제권이 없음. - 여러 문제 발생 가능(업데이트 문제, 지원 문제, 라이센스 문제)

2. Process Activities

specification - development -development - validation - evolution

Specification
-요구 사항 추출
-요구 사항 정의
-요구 사항 검증

software design - 명세서를 realise하여 구조를 구축하는 것
software implementation - 해당 구조를 실제 실행가능 프로그램으로 바꾸는 것
옛날에는 이게 명확히 구분되었지만 요즘날은 이 두 개의 경계가 얇아지고 서로 밀접한 관계를 지니게 됨.
(Agile 기법을 적용하게 되면서 디자인과 구현이 한번에 이루어지는 경우가 많아짐)

Testing Stages
1. Component testing
 - 각 개별적인 요소들은 독립적으로 테스트
2. System testing
 - 모든 요소들을 한번에 시스템으로 합쳐서 테스트
3. Customer testing
 - 실제 고객들이 사용하면서 테스트 (요구사항이 충족되었는지 테스트 함)

Plan-driven testing phases (V-model)
어떻게 테스트할 것인지 미리 문서로 다 작성.
ex) 이 테스트 시나리오대로 테스트하고 요구 사항이 충족되는지 확인해주세요.
Acceptance test plan - System Integration test plan - Sub-system integration test

Software evolution
소프트웨어는 유연성이 있으며 변화 가능해야 함.
비즈니스 상황이 바뀌면서 요구 사항이 바뀔 수 있음. 이를 대응할 수 있어야 함.
유지보수 및 확장성이 매우 중요하다!

3. Coping with change

요즘날 변화는 불가피한 것이다.
변화를 피하려고 하는 것이 아닌 변화를 준비해야 한다.
변화하는데 드는 비용을 줄이는 방안으로 소프트웨어 구상

Reducing the costs of rework(Prototype 개발, 점진적인 개발(version별로 개발))
Change anticipation - 변화를 미리 예상
ex) prototype 개발 - 주요한 기능들을 고객들에게 미리 보여주고 반응을 보고 개발.
Change tolerance - 변화 비용이 적게
ex) Incremental development 방법으로 변화에 대응하기.
이전 버전의 반응들을 토대로 이후 버전에 반영하여 변화에 대응하기.

System Prototyping
주요 기능을 빠르게 개발하여 고객들의 반응을 보면서 요구사항 검증
요구 사항 추출 및 검증에 도움을 준다.
이점
1. 시스템 사용성 증가
2. 실제 유저들의 니즈들을 최대한 가깝게 만족시킬 수 있음.
3. 디자인 퀄리티가 증가
4. 유지보수성 증가
5. 개발 비용 줄어듬(모호했던 부분들을 미리 해결함으로써 실제 개발할 때 비용 줄어듬.)
- 모호했던 것들이 확실해지면서 여러 부분에서 이점을 볼 수 있음.

중요 원칙 - 프로토 타입은 버려야 한다!

프로토 타입은 유지보수 생각을 하지 않고 기능에 초점을 맞추어 개발함.
-> 이를 그대로 사용할 시 이후 스파게티식 코드로 인해 유지보수에 많은 문제 발생가능.
- 또한 프로토 타입은 문서화 되지 않았음.
- non-functional 요구사항을 고려하지 않음.
- 프로젝트의 질이 전체적으로 떨어짐(빠른 기간안에 개발함.)
-

Incremental delivery
이전 버전의 반응들을 토대로 이후 버전에 반영하여 변화 대응.
이전 버전 배포들을 통해 유저들의 요구 사항 추출 가능. 이후 버전에 반영
프로젝트 실패 위험이 적음.
문제점
1. 대부분 시스템은 기본적인 기능들이 다른 부분들의 기능을 조금씩 필요로 함. -> 서로 다른 컴포넌트를 조금씩 쪼개서 개발하다보니 나중에 합치기 어려울 수 있음.
유지보수 관점에서도 안 좋아질 수 있음.

2. 라이센스, 외부 제약에 맞춰야 함.

4. Process improvement

product quality 증대, 개발 비용 감소시키는 방향으로 Process 개선시키는 것

1. Process measurement
- 정량적으로 측정할 부분 찾기.
- 프로세스 측정요소
특정 process activity 걸리는 시간, 특정 활동에 요구되는 자원량, 특정 이벤트가 발생되는 횟수 등등
CMMI(능력 성숙도 통합 모델)
- Initial, Repeatable, Define, Managed, Optimising(SEI capability)
2. Process analysis
- 평가 및 분석하기.
- 현재 프로세스의 약점 식별.
3. process change
- 식별된 현재 약점을 개선시킬 수 있는 방향으로 프로세스 변화
- ex) 좀 더 효율적인 방식으로 data collecting

Chap 3 - Agile Software Development

- 이전 소프트웨어들 - = 소프트웨어를 팔아서 이득을 취함(오픈 소스 X)

마이크로소프트 윈도우 & 오피스
한컴 한글 & 워드 프로세서
어도브 포토샵
=> 큰 규모의 소프트웨어 개발 - 오랜 기간 개발
- 사용자의 요구 반영 X -> 일단 계획하고 끝까지 개발하고 오랜 시간 끝에 출시.
- Hard deadline 가짐.
- 엄청난 양의 재화와 인력이 들어감.

Plan-driven & WaterFall 방식의 개발
1. 관리를 체계적으로 할 수 있음.
2. 개발 시작 전에 이미 많은 부분을 고려하고 고민을 했기 때문에 이후 발생하는 문제가 적음.
3. 모든 단계들이 문서화가 되어있기에 새로운 멤버가 들어와도 빠르게 적응할 수 있음.
4. miscommunication으로 인한 오류가 적음. 모든 것이 문서에 자세하게 명시되어 있기 때문.

문제점
1. 오랜 전에 기획한 내용들이 현재에는 유효하지 않을 수 있음. 따라서 변화가 요구되는데 변화를 적용하기 위해선 다시 처음 단계로 돌아가야 하는 큰 비용이 발생함.
2. 완벽하게 이전 단계를 끝내고 다음 단계로 가는 경우가 사실상 불가능함
- 이전 단계가 다음 단계 경험으로부터 도움을 받을 수 있음.
- 이후 단계를 진행하면서 이전 단계가 명확해질 수 있으며 좀 더 좋은 개발이 가능할 수 있음.
- 개발 도중에 요구 사항 변경 or 비즈니스 시장 변화 등으로 인해 소프트웨어 변화가 필요할 경우 다시 처음으로 돌아가 시작해야 함.

- 요즘 소프트웨어들- 서비스를 이루기 위한 도구적 측면의 소프트웨어들

구글 & 네이버 검색 서비스
다음, 카카오, 라인 메신저
광고 베이스들 서비스들
=> 빠른 기간 안에 개발 출시 - 이후 유저들의 반응들을 보고 요구 사항 수정
= 가능한 한 빠르게 출시.
= 제한된 재화와 인력들
= 시장 반응에 빠르게 반응

What is Agile

Agile Manifesto - 4대 원칙

1. 개인과 상호작용
2. 작동하는 소프트웨어
3. 고객과 협력
4. 변화에 대응

12 Principles of Agile Development

원래 여기서 더 규칙을 안 만들려고 했는데 개발자들이 좀 더 상세한 규칙을 필요로 하다고 해서 나온 12개 원칙

1. 고객 만족을 최우선으로 한다.
2. 요구 사항 변화에 인색하지 말고 환영하라.
3. 작동하는 소프트웨어를 자주, 빈번하게 배포하라
4. 사업가와 개발자가 같이 일해라. (다른 부서랑도 소통해라)
5. 각 개인이 프로젝트 개발에 동기부여가 되어야 한다. (팀원들이 열심히 프로젝트에 참여)
6. 직접 얼굴을 보면 대화하라(화상회의도 괜찮) - 가장 좋은 정보 전달 방법
7. 작동하는 소프트웨어가 진행 정도를 측정하는 주요 척도이다. 작동하는 소프트웨어로 진도 측정
8. 지속가능한 개발을 추구하라. 지속 가능한 개발 속도 유지.
9. 좋은 설계, 좋은 기술에 계속해서 관심을 기울이자. 자기계발을 게을리하지 말자.
10. 단순성을 최대화하자. (효율성을 증대하라)
11. 최고의 구조나 디자인, 아키텍처는 스스로 수행하려는 팀에서 나온다
12. 정기적으로 팀은 조금 더 효율적인 방안을 모색하고 팀의 방향을 맞춰야 한다.

Scrum

Agile은 철학이다. 이 Agile로부터 나온 방법론이 바로 Scrum이라고 생각하면 된다.

=> 개발을 반복 가능한 time box로 나누고 실행.

=> 1-4주 Sprinit +15min daily 회의

- 1~4주 동안 개발 주기마다 실제 동작하는 소프트웨어 배포
- 매일 15분 정도의 meeting 통해 프로젝트 진행 (자유로운 발언 가능)

추구가치 5가지

1. 용기 - 옳은 일을 할 수 있도록 팀원과 갈등, 도전을 위한 용기를 가져라!
2. 집중 - 할 일에 집중해라. (불필요한 회의, 불필요한 루틴들 제거)
3. 헌신/책임 - 자기가 맡은 분야에 책임을 갖고 완수하라.
4. 존중 - 팀원들을 존중해라
5. 투명성/개방성 - 프로젝트에 대한 모든 내용을 투명하게 공개하라.

(불리한 내용을 숨기지 마라.)

Product Owner
- 제품 요구사항(백로그) 관리 및 설명
- 제품 백로그 우선순위 관리
- 요구사항 충족 확인
Scrum Master
- 팀의 갈등 조율 및 장애 요소 해결
- daily meeting 진행
- 모니터링 및 Tracking

DevOps
변화를 시스템에 적용하고 배포하는 시간을 줄이기 위한 방법
자동화가 키워드다!
소프트웨어 개발의 모든 단계를 모니터하고 자동화시키는 것!

소프트웨어 개발팀과 운영팀이 서로 자주 커뮤니케이션을 하며 빠른 시간에 서비스를 개발 및 배포하는 것이 목적이다!

제품 출시까지 걸리는 시간 단축
여러 자동화 기능 툴을 이용함.
빌드 -> 자동화(통화 & 배포) = 빠르게 출시 가능.

Summary

1. **Agile Manifesto**
2. **12 Principles of Agile Developments**
3. **Scrum**
4. **DevOps**

Chap 4 - Requirements Engineering

Functional and non-functional requirment
requirement engineering processes
추출, 정의, 타당성 검증, 변화/개선/유지보수

Requirement enginnering = 고객/유저의 요구사항 과 제한사항을 정의하는 것.
requirement 자체가 계약이 될 수 있으며 매우 자세히 다뤄져야 한다.
이게 개발의 시작이고 근본이다. - 최초의 고객/계약자과 개발자의 합의서이다.

Types of requirement
1. User requirements
- 고객을 위한 요구사항 정의이다. 자연어와 다이어그램 등을 이용해 고객들이 쉽게 이해할 수 있게 표현한다.

2. System requirements
실제 개발자들이 보게되는 requirement이다. 개발자를 위해 좀 더 구체적으로 각 기능을 정의했다.
[Chap4 7Page에 멘탈케어 시스템으로 예시가 잘 나옴.]

Agile 방법에서 requirement 관점
Agile 기법에서 너무 자세한 requirement 문서는 시간 낭비라고 여김.
결국 요구 사항은 계속해서 변하기 때문에 이를 자세하게 문서로 정의하지 않음.

1. Functional and non-functional requirements

Functional Requirement - 기능 구현에 중점
시스템이 실제로 제공해야 되는 서비스들을 의미.
실제로 시스템이 어떻게 동작해야 하는지 서술되어 있음.
서비스 동작에 관련하여 여러 가지 시나리오로 기술되어 있음.(자세하게 매우 세세하게)
멘탈케어 시스템 예시) 유저는 모든 클리닉에 대한 일정 리스트를 검색할 수 있어야 한다.
시스템은 매일 어떤 환자가 방문할지 리스트로 보여줘야 한다.
각각의 직원들은 8자리의 고유 번호로 등록되어 있어야 한다.

Requirements imprecision - 요구 사항 부정확
기능 요구 사항이 제대로 명시되어 있지 않거나 애매한 부분이 있을 경우 문제가 발생한다.
모호한 요구 사항은 개발자와 유저/계약자 사이끼리 서로 다르게 해석하여 문제가 발생할 수 있다.
예시) 유저의 해석 - 모든 지점의 appointment 검색 가능
개발자의 해석 - 각 지점의 appointment만 검색 가능

이런 문제를 해결하기 위해선 Complete와 Consistent가 지켜져야 한다!

Complete - 모든 요구 사항이 명확하게 정확하게 명시되어야 한다.
Consistent - 요구 사항들이 서로 충돌되지 않게 일관되게 작성되어야 한다.
하지만 실제 실무에서 이는 매우 어려운 과제!!!!
모든 요구 사항 명세서를 정확하게 만드는 것은 사실상 불가능!

Non-functional Requirment - 기능 외에 것들에 대해 중점
ex) 서비스 제약 사항들, 규격들, 규범들, 표준 사항들, domain requirement 등등
신뢰도, 반응시간, 필요한 저장공간 크기 등등 기능 외적인 요소들

Non-functional Requirement는 보통 기능적인 요소들보다 더욱 치명적이고 만약 이것이 안 지켜질 경우 해당 시스템이 쓸모가 없을 수 있다.
비기능 요구사항은 해당 시스템 아키텍처 전반에 영향을 끼친다!!
예를 들어) 성능에 대한 요구사항이 만족되려면 - 시스템의 component 간의 communication 수를 줄여야 한다.

Non-functional classification(비기능 명세)
1. Product requirement
- Product가 요구한 대로 작동해야 한다. (실행가능, 속도, 신뢰성 등등)
예시) 멘탈케어 시스템은 5초 이상 Down되면 안된다.
2. Organisational requirements
- 조직의 정책과 절차를 만족시켜야 한다. (조직의 규칙, 규범, 구현 조건 등)
예시) 멘탈케어 인증 시스템은 identity card로 이루어진다.
3. External requirements
- 시스템 외부에서 발생한 요구사항들이다. (ex. 법적인 문제, 라이센스 문제, 정치적 문제)
예시) 멘털케어 시스템은 환자의 프라이버시를 보호해야 한다.

Non functional 요구 사항 검증의 어려움(정량적으로 측정하기 힘든 추상적인 경우가 많다)
요구 사항을 정확하게 명시하는 것은 매우 어려운 일이다.
당연히 이를 검증하는 것도 쉽지 않다.
검증 가능한 목표를 설계하고 이를 통해 검증해라.

1. 고객이 추상적인 요구사항을 요구하면 이를 정량적으로 검증 가능한 방법으로 바꿔라!
ex) Speed - screen에 표시되는 시간, transaction이 걸리는 시간으로 검증
Easy Use - 에러가 적어야 한다 -> 4시간 직원 교육을 통한 실수 줄이기.
그 외 요소에 대한 측정 방법은 (Chap4 p.26 참고)

2. Requirements engineering process

elicitation, analysis, validation, management

Requirements elicitation (요구사항 추출)

요구 사항 추출 or 요구 사항 도출 이라고 부른다.

1. 요구사항 도출
- 개발자와 이해관계자끼리 서로 상호작용하여 요구사항 도출하는 것.
2. 요구사항 정리 및 분류
3. 우선순위 부여 및 타협(협상)
4. 명세화 - 각 요구사항들을 문서화하는 것.

Problem (요구사항 도출할 때 생기는 문제점들)

1. 이해관계자, 주주들이 실제로 자기 뭘 원하는지 잘 모르는 경우가 있음.
2. 주주들이 그들만의 단어와 용어로 이야기함. 이해하기 힘듦.
3. 서로 모순되는 요구사항이 도출되기도 함.
4. 회사 규율 또는 정치적인 요소가 요구사항에 영향을 끼침
5. 요구사항이 도중에 바뀜.

요구사항 도출 방법들

Interview 방식을 통해 요구사항 도출하기
closed Interview - 미리 질문지를 작성해가서 인터뷰
open Interview - 다양한 주제로 자유롭게 이야기하며 요구사항 도출하기
효과적으로 인터뷰를 하기 위해선 Open mind로 선입견을 가지지말고 이야기 할 것!!!

인터뷰의 문제점
그들만의 용어와 단어로 이야기하여 요구사항 엔지니어가 이를 이해하기 힘든 경우가 있음.
몇 개의 domain 지식이 요구사항 엔지니어한테 중요하게 안 보일 수 있음.

Ethnography - 민족지학 관점에서 실제 어떻게 쓰이는 지 연구함

실제로 사람들이 어떻게 일하는지 분석
-> 사람들은 자신들의 일을 설명할 필요 없음.
중요한 것은 해당 일을 관찰하는 것임.
시스템을 단순히 모델로 표현한 것보다 실제 작업은 훨씬 더 복잡하고 어려움.

즉, Ethnography 관점에서 요구사항 도출을 살펴보면

실제로 작동하는 방식으로 요구사항이 도출되지 않고 그들이 원하는 방식으로 도출됨.
다른 사람과 협력하면서 그 사람의 일을 인지하는 과정 속에서 요구사항이 도출됨.

Prototype을 제작해서 실제 사용하는 것을 지켜보면서 요구 사항을 도출해라.

스토리와 시나리오 - 사람들이 실제로 어떻게 사용할지 예시를 들어 설명하는 것.
-> 시스템이 특정한 경우에 어떻게 사용되고 동작되는지 명시하는 것.
이 예시는 Chap4 page46~ 에 나와있음.(iLearn 예시)

3. Requirements specification

요구사항 정의 & 문서화

User requirment - end-user가 이해할 수 있게 작성된 문서
System requirements - 개발자가 실제로 보는 문서 (매우 자세하게 각 시나리오 별로 어떻게 작동해야하는 지, 어떤 요구 사항을 충족시켜야 하는지 적혀있음.)

System 요구사항 문서화 방법들
1. 자연어 - 우리 말로 풀어서 설명
문제점 -> 명확하지 않다. 말로 설명하다 보면 애매한 부분이 생긴다.
요구사항에 혼란이 올 수 있다. (기능적/비기능적 요구사항이 같이 쓰일 수 있다.)
여러 개의 다른 요구 사항이 섞여서 표현될 수 있다.

2. Structured 자연어 - Sudo 코드 사용

3. Design description languages - 프로그래밍 언어를 사용하여 표현

4. Graphical notations - UML 사용하여 표현
Use-case = Actor와 상호작용으로 표현(멘탈케어 예시 p.64)
5. Mathematical specifications - 유한 상태 머신 사용(finite-state machines)

Requirement and Design (매우 밀접하게 연관되어 있다.)
이 두 개는 서로 분리해서 보기보단 밀접하게 연관시켜 봐야 한다.

4. Requirements validation

요구사항 검증(제대로 충족되었는지 검사)

시스템이 진짜 고객이 원하는 대로 동작하는지 검증하는 단계
요구사항이 잘못 정의되어 발생하는 error cost는 막대하다. 따라서 이 요구사항 검증단계는 매우 중요!!!
구현한 뒤에 수정하는 것 = 요구사항 수정 * 10000000000

1. Validity - 시스템이 제공하는 기능이 고객의 요구사항을 충족시키는가?
2. Consistency - 어떠한 요구사항도 서로 충돌하지 않는가?
3. Completeness - 모든 기능들이 고객이 필요로 해서 작성된 것이 맞는가?
4. Realism - 작성된 요구사항들이 사용가능한 예산과 기술들로 실제로 구현이 가능한가?
5. Verifiability - 요구사항들 검증이 가능한가?(구현한 뒤 실제로 정량적 측정이 가능한가)

validation techniques

1. 요구사항 review - 다시 한번 검토하기 (개발자와 계약자/운영팀 등 함께 검토를 진행해야 함) 오해한 부분이 없는지 의도한 대로 요구사항이 작성되었는지 검토

Review할 때 check할 사항들

[Verifiability(검증가능한지), Comprehensibility(누구나 같은 이해를 하는지),
Traceability(추적이 가능한지 - 각 문서에 번호를 매겨서 어디에 어떤 요구사항이 있는지 찾을 수 있어야 함.),
Adaptability(추후에 요구사항이 바뀐다고 해도 다른 요소들에 영향이 없는지)]

2. Prototype - 프로토 타입 개발하여 요구사항 검토하기
3. Test-case generation - 테스트 케이스를 생성하여 가상으로 돌려보기.
test case를 작성한 뒤 이를 머리 속 or 종이 같은 걸로 한번 검증해보기.

4. Requirements Change

요구사항 변화

비즈니스적 상황이나 기술적 환경은 언제든지 변화할 수 있음.
ex) 새로운 핸드폰이 출시됨(안드로이드 버전도 바뀌고 화면 크기도 바뀜)
새로운 법/제도가 도입됨.

요구사항 변화를 받아들여야 하는지 결정해야 함.

문제 식별 -> 문제 분석 -> 발생되는 비용 분석 -> 구현 바꾸기 -> 요구사항 수정

Chap 5 - System Modeling

Context Model, Interaction Model, Structural Model,
Behavioral models, Model-driven engineerin

System modeling이란?
시스템의 추상적인 모델을 개발하는 것을 의미한다.
각 모델은 시스템의 다른 관점,View를 의미한다.

시스템 모델링은 그래픽으로 표현되는 경우가 많고 대표적인 예시가 UML이다.
모델은 분석가들이 시스템의 각 기능을 이해하기 쉽도록 도와주며 고객과 원활한 의사소통할 때에도 매우 유용하게 사용된다.

존재하는 시스템을 모델링하는 것 -> 설명/대화를 위한 도구로 사용
실제 존재하는 시스템이 무엇을 하는지 명확하게 보여줌
새로운 시스템을 모델링하는 것 -> 시스템 이해관계자들에게 요구사항을 설명하는데 도움을 줌.
model-driven engineering -> 모델을 실제 product로 구현 가능.
하지만 교수님은 이를 추천하지 않는다고 함.
modeling 할 때는 미처 발견하지 못 했던 문제가 구현 할 때 발생가능.
그리고 모델은 실제와 차이가 있을 수 있으며 이를 원하는 대로 정확하게 구현하는 프레임워크 존재하지 않는다.

System perspectives
5. external perspective(외부관점)
- 외부 관점에서 크게 보는 것(시스템 환경이나 context 관점)
6. interaction perspective
- 각 시스템 사이의 상호작용하는 중점의 관점
7. structural perspective
- data가 시스템에서 처리되는 과정 중점의 관점
8. behavior perspective
- 각 이벤트에 어떻게 행동하는지 행동 중점의 관점

UML diagram types

9. Activity diagram - process, data processing 위주
10. Use case - 시스템과 환경 상호작용 위주
11. Sequence - 각 시스템과 컴포넌트 상호작용 위주
12. Class - 오브젝트 클래스 위주
13. State - 각 이벤트에 대한 react 위주

graphical model 사용의 이점(중요)

1. 시스템에 대한 의논할 때 주요한 수단이 될 수 있음.
부정확하거나 완벽한 모델이 아니어도 괜찮음. 그저 discussion 하는데 도움을 주면 OK
2. Documeting의 한 방법으로 사용될 수 있음.
model은 시스템을 도식적으로 보여주는 것.
3. 자세한 설명이 추가되어 시스템 구현하는데 도움을 줄 수 있음.

Context Models

추상적인 상위 레벨 모델링

이들은 매우 추상적인 단계의 모델로 무엇이 시스템이 포함되는지 포함되지 않는지 구분하는 역할을 함. 즉, System boundaries를 보여줌
+다른 시스템들과의 관계를 보여줌

Context Model은 단순히 다른 시스템들과의 관계나 환경을 보여줌. 어떻게 개발할지에 대한 내용은 들어있지 않음.

Interaction Models

시스템들간의 상호작용을 보여주는 모델

Use case / sequence diagram을 통해 interaction을 보여줌.
유저의 상호작용을 모델링. 시스템 간의 상호작용을 모델링.

Use Case Modeling
요구사항 도출을 위해 사용되기도 했고 지금은 UML로 통합됨.
각 use case는 구분된 경우로 보여줌.
Actor - user or other system
ex)
[Actor](Medical receptionist) - > Transfer Data -> [Actor]Patient record System.
관리자가 record system으로 데이터 전송하는 내용을 나타내는 Use case
Tabular description - 표로 좀 더 자세하게 내용을 정리하여 나타내는 방법.

Sequence diagrams
UML의 한 종류이다.
actor 간의 obejct 간의 상호작용을 나타내는데 주로 사용된다.
interaction이 일어나는 과정들을 자세하게 보여줌.[Chap5 p.21 참고]

Structural Models

시스템의 조직도/관계도를 보여주는 모델

System architecture designing에 대해 고려하고 있을 때 사용.

Class Diagram
각 Class가 어떻게 Mapping되는지 보여줌
Consultation class
내가 흔히 아는 Diagram.

Generalization 일반화 (중복성 제거/ 재사용성 증가)
각 Object들의 공통된 부분을 한 곳에 묶어서 저장.
base class - 추상적/공통
sub class - 구체적/특화

Structural Models

시스템의 다양한 behavior을 보여주는 Model
특정 이벤트에 어떻게 react 하는지 보여줌.

많은 business system은 data-processing system.
즉, data가 매우 주요하게 다뤄지는 시스템들임.
그들이 어떻게 input되고 어떻게 process되는지가 중요

Data-driven model은 data input부터 generating부터 output까지 자세히 보여줌.
Event-driven model - 내부적/외부적 event에 어떻게 반응하지 자세히 보여줌.

State machine models - 유한 상태 기계(이거랑 비슷할 듯)
시스템이 외부/내부 사건에 어떻게 반응하는지 보여줌.
real-time system에 자주 쓰임. 인슐린 주입 시스템(제때에 event 발생하는 것이 매우 중요)

Model-driven engineering

교수님이 크게 말할 것 없다고 하심.

그냥 이런 게 있다는 것만 알라고 하심.

P.48에 장점과 단점이 있음. 필요하면 볼것

Model ->Program (자동 구현)
모든 개발자가 원하는 환상적인 프로그램.
하지만 이게 현실적으로 가능하지는 모름.
주변에 사례도 없고, 실패 사례만 가득함.

Chap 6 - Architectural Design

Architectural design decision, Architectural view,
Architectural pattern, Application architectures

Architectural design - 시스템이 어떻게 조직되었는지 이해하는 것이 주요 관심사
이후 시스템의 전반적인 구조를 desigining

시스템 아키텍처를 이후에 Refactoring 하는 것은 비용이 매우 비싸다.
그래서 처음에 제대로 design 하는 것이 중요하다.

시스템 아키텍처 explicit의 장점
1. 주주들과 소통하기 편하다. 시스템이 실제로 어떻게 연결되어 있는지 보여주면서 설명 가능
2. 시스템 분석하는데 용이하다.
3. 유사한 요구사항으로 정의된 시스템을 재사용할 수 있다.
software reuse 도와줌. 즉, 소프트웨어 재사용성이 증가한다.

매우 추상적이다. 각 component간의 관계를 자세히 설명하지 않음. 하지만 다른 부서들과 소통할 때, 설명할 때 큰 도움을 줌.

Use of architectural model - 아키텍처 모델 사용법

14. system design 논의할 때 용이하게 사용
15. 디자인된 아키텍처를 문서화 할 때도 용이하게 사용

Architectural design decisions

시스템 아키텍처 디자인할 때 할만한 질문들(p.13)
16. 어떤 Architectural pattern을 사용했는가?
17. 아키텍처가 문서화 되었는가?
18. 근본적인 접근 방법이 무엇인가

Architecture reuse

시스템 안에는 유사한 도메인이 있고 유사한 아키텍쳐를 지니게 됨. 따라서 재사용성 증가!
이미 수많은 선배 개발자들이 시행착오를 겪고 이를 패턴화 시킨 것들이 있음.
이를 공부하고 상황에 맞게 재사용하자!!

Architecture과 시스템의 특징들

1. Performance
 - communication을 최소화하자.
2. Security - inner layout 사용하자
 - 중요한 내용들이 밖에서 안 보이게 감싸자!
3. Safety
 - 안정성 추구
4. Availability
 - 예비 장비를 두어 죽지 않고 프로그램이 돌아가게 하자.
5. Mainability(유지보수성)
 - 추후에 component 교체/유지보수가 쉽게 하자.

Architectural views

어떤 View랑 Perspective가 시스템 아키텍쳐 디자인할 때 유용할까?
어떤 notations(표기법)이 사용될까?

Logical View - object, class들을 보여주는 View
Physical View - 시스템 하드웨어적 측면을 보여주는 View
Development View - 시스템의 각 component들이 개발할 때 어떻게 나뉘어는지 보여줌.
Process View - run time에 각 process가 어떻게 상호작용 보여주는 view
+ 1 => use case or 시나리오

각 회사 규율에 맞게 회사에 쓰는 방식을 따를 것!

Architectural pattern

key word : reuse/share

이미 선배들이 겪은 패턴화된 지식을 재사용/공유하는 것
이미 다양한 환경 속에서 시도하고 테스트한 여러 좋은 design 패턴이 있음. 이를 재사용!

유명한 Design Pattern MVC(Model-View-Controller)

소프트웨 개발을 할 때 Model - View - Controller로 나누어서 개발하는 디자인 패턴.
쉽게 유지보수 가능.
세 부분으로 나누어서 개발하므로 이해하기 쉽고 의존성이 낮아진다.
특정 로직 코드에만 집중할 수 있다.
각 Component에 영향을 주지 않고 수정가능(Display 영향 없이 수정 가능, 로직 영향 주지 않고 View 수정 가능) But 추가적인 코드와 코드 복잡성이 증가할 수 있다.(서로 상호작용)

Layered architecture (계층적 구조)
여러 개의 Layer로 불리. ex) API 7계층
layer가 수정될 때 오직 인접한 Layer들만 영향을 받음.

Repository architecture
Center database가 존재하고 sub-system들이 이에 접근함. => data share
많은 양의 데이터가 센터에서 공유됨 -> 효율적인 데이터 공유 메커니즘임.

Client-server architecture
분산된 시스템 모델이다. - 넓게 흩어진 component들에서 data가 어떻게 처리되는지 보여준다.
여러 개의 서버들이 특정한 서비스를 Network를 통해 제공한다.
Network를 통해 Client가 Server에 접근하고 Server는 유저가 요구하는 특정 Service를 제공한다.
Youtube, gmail, Cloud system, cloud storage

Pipe and filter architecture
input이 발생하면 순차적으로 처리 후 output 발생
순차적으로 일어나는 Batch sequential model.
interactive system에 적합하지 않음.
https://hodev.tistory.com/60?category=966958 **-> 위의 5가지 디자인 패턴 중요!!**

Application architecture

business 문제들은 많은 공통된 요소들이 존재하다.
이 공통된 요소들을 한데 묶어서 정리하여 재사용하는 것이 중요!!
공통된 부분을 가져다가 쓰고 그 외 특정 상황에 맞게 specific한 부분을 추가.

Use of application architecture (어플리케이션 아키텍쳐의 사용)
19. 이전 아키텍처를 토대로 내 아키텍처 디자인을 검토/검증 가능.
20. 각 작업들을 조직화하고 조율할 수 있음.
21. 코드의 재활용성 증가
22. communication의 도구로 사용될 수 있음.

application type
23. data processing applications - 데이터 처리가 순차적으로 일어남. 중간에 유저의 개입이 없음.
24. transaction processing application - 데이터 센터가 존재하며 유저가 요청하며 요청에 맞게 데이터를 수정함. e-commerce, 예약 시스템
25. event processing system
26. language processing system - 컴파일러, interpreter

Web-based information system
- 요즘 대부분 서비스들은 Web-based로 작동함. 즉 Networking이 필수다.

그래서 Server 구현이 매우 주요한 과제이다.
모든 User의 communication을 책임지고 담당해야 한다.
information을 data로 변환하여 database에 잘 저장해야 한다.

Key point

27. 소프트웨어 아키텍처는 어떻게 시스템을 조직할지를 묘사한 것이다.
28. 아키텍처는 여러 관점에서 문서화될 수 있다.
29. Architecture patter은 일반적인 시스템 아키텍처에 대한 지식을 재사용하는 것이다.
30. 아키텍처의 사용은 우리가 이해하기 쉽게 도와준다.
애플리케이션 검증 및 비교 분석, 큰 규모의 컴포넌트 재사용까지...
5. Transaction system은 시스템의 상호작용이다. 어떤? 데이터베이스에 정보가 드나드는.
6. Language processing은 한 언어를 다른 언어로 번역할 때 사용된다.
입력된 언어에 대한 특별한 조치를 취한다.

Chap 7 - Design and Implementation

Object-oriented design using the UML, Design patterns
Implementation issues, Open source development

software design은 창의적인 활동
고객의 요구사항에 기반한 컴포넌트 간의 관계를 식별하는 활동!

Build or Buy
COTS 빠질 수 없지! - 개발되어 있는 프로그램을 사서 사용
예를 들어 너가 Medical Record System을 개발하려고 해.
이때 그냥 병원 측에서 패캐지를 하나 사고 이를 이용하면 좀 더 저렴하고 빠르게 개발 가능.

software를 사서 사용하면 고려해야 할 점은 2개!
2. 어떻게 세부적인 설정을 할 건지
3. 어떻게 요구사항을 충족시킬건지

Object-oriented design using the UML

객체지향 디자인 방법
이 방법은 개발할 때 많은 노력을 필요로 함. - 만약 작은 시스템 개발을 한다면 이는 효율적인 방법이 아닐 수 있음.
그러나 큰 시스템을 개발할 때 이 방법은 매우 유용한 방법 - 유지보수 측면에서 유용

Design patterns

코드만 재사용하는 것이 아닌 디자인도 재사용하자!
디자인 패턴은 이미 정형화된 문제에 대한 어르신들의 노하우/솔루션이다!

예시들
1. Observer - display와 object를 분리하여 개발하는 패턴
한 Object에 대한 여러 display (한 오브젝트가 표로도 표시되고, 차트로도 표시되고 그림으로도 표시되는 등 Multi display가 있을 때 유용)
Object에서 변화가 생김 -> 연결된 여러 개 display에게 알려서 변화주기.

디자인 패턴 문제점
디자인 패턴을 사용하기 위해선, 문제가 발생했을 때 해당 문제가 무엇인지 너가 정확히 알고 있어야 하면 이에 적용할 수 있는 패턴도 인지하고 있어야 함.
4. Observer Pattern - 한 오브젝트에 대한 multi display 존재
5. Facade pattern - 인터페이스 정리(연관된 여러 오브젝트) 점진적으로 개발될 때
여러 클래스와 컴포넌트가 개발되고 소프트웨어가 커짐에 따라 이런 연결/연관 관계가 매우 복잡하게 얽히게 됨. 이는 실수를 유발하고 결국 유지보수 비용이 커짐. 하지만 이런 여러 인터페이스들을 정리하여 좀 더 단순화한 인터페이스를 제공하여 복잡하게 얽힌 관계를 풀어내는 것이 바로 Facade patter이라고 볼 수 있다.
6. Iterator pattern - collection 접근에 대한 표준 방법
7. Decorator pattern - run time에 클래스의 기능 확장.

Implementation issues

1. Reuse - 가능한 이미 존재하는 컴포넌트/시스템은 재사용해야 한다.
2. Configuration management - 개발이 진행되는 동안 배포되는 여러 Version들을 Tracking 할 수 있어야 한다. Version 관리가 매우 중요
3. Host-target development - 너는 한 개의 단일 컴퓨터에서 개발되지만, 사용될 때는 여러 다른 컴퓨터에서 사용됨. 이를 고려하고 다른 매체/디바이스에서도 잘 돌아가게끔 구현해야 함.

Reuse
8. abstraction level - Architecture and design pattern
9. object level - Programming language libraries. 라이브러리
10. component level - framework
11. system level - COTS(그냥 앱 자체를 사서 재사용)

재사용한다고 비용이 안 드는 것은 아니다.

12. 나에게 적절한 software를 찾는 비용
13. 해당 시스템/소프트웨어를 구입하는데 드는 비용
14. 이를 내 목적에 맞게 설정하고 바꾸는데 드는 비용
15. 이를 기존에 내가 개발하던 것과 통합시키는데 드는 비용

Configuration management
- 소프트웨어 시스템 변화 관리
- 형상관리툴 생각
- 어떤 commit이 일어났는지 어떤 코드가 변화했는지 추적할 수 있어야 함.

Open source development

Free Software Foundation - Open source를 옹호함.
source code는 개인 소유나 재산이 아닌 오히려 모든 유저가 이용할 수 있어야 하면 이를 평가하고 그들이 원하는대로 수정이 가능해야 함.

오픈소스 소프트웨어는 인터넷을 통해 많은 자원 개발자가 서로 소프트웨어를 수정해고 발전시키면서 널리 퍼지게 됨.

Linux OS - open source 대표적인 예시
Java, Apache Web server, MySQL database 등등

REUSE 증가
점점 더 많은 회사들이 오픈소스 접근법을 사용하면서 개발함.
그들의 비즈니스 모델은 소프트웨어 파는 것이 아닌 서비스를 파는 것.
소프트웨어는 그저 도구적인 관점으로 전략.
더 저렴하게, 더 빠르게 소프트웨어 개발이 가능해짐.

Open source Licensing

오픈소스의 근본적인 개념은 자유롭게 사용가능이다.
이는 아무나 이 코드를 상업적으로 이용해도 된다는 것이 아니다.

법적으로 코드의 소유는 해당 개발자에게 있다. 그들이 코드 사용에 제한을 걸 수 있다.
GNU - 너가 만약 이 소스를 이용해서 개발했다면 그 개발한 소프트웨어도 오픈 소스로 공개하라.
LGPL - 코드의 수정이 사용할 경우 이 코드를 이용했다고 명시만 하면 상업적으로 배포해도 상관 없음. 그러나 코드의 수정이 조금이라도 있을 경우 오픈소스로 공개해야 한다.
BSD - 아무런 제약 조건 없음. 너가 원하는 대로 해라.