8장 소프트웨어 테스팅 Software testing

1. 테스팅의 목적

- 프로그램이 의도적으로 수행되는지 보여줌

- 프로그램 사용 전에 결함이 발견

2. 프로그램 테스팅 Program testing

- 인위적인 데이터 이용

- 프로그램 실행

- 실행 결과를 점검

-> 테스팅은 오류 존재를 밝힐 수 있지만, 프로젝트 자체에 오류가 없음을 보일 수 없다.

3. 소프트웨어 테스트 시, 2가지 수행 > 영어도 같이 외우기

*릴리스 : 마지막 제품이 될 가능성이 있는 배타버전, 상당히 버그가 나타나지않으면 출시할 준비가 되었음을 의미

1. 검증 테스팅 validation testing

- 소프트웨어가 고객의 요구사항에 맞는지 보여줌

- 맞춤 소프트웨어의 경우

  요구사항마다 적어도 하나의 테스트가 있어야 하며 시스템이 정확하게 수행되는 것을 기대

- 일반 소프트웨어의 경우

  제품 릴리스에 포함될 모든 시스템 기능들을 위한 테스트가 있어야 함 / 기능들의 조합도 테스트

2. 결함 테스팅 defect testing

- 소프트웨어의 결함(버그)에 의해 제대로 동작하지 않는 경우를 찾음

- 테스트 케이스는 결함을 드러낼 수 있도록 설계

4. 소프트웨어 검증 및 확인 (V&V) 프로세스 > 영어로 외우기, 시험

테스팅은 폭넓은 소프트웨어 검증 및 확인 프로세스의 일부

1. 검증 Verification

- 제품을 올바르게 만들고 있는가?
- 소프트웨어가 요구사항에 맞는지 점검한다.
- 소프트웨어가 기능적, 비기능적 요구사항 명세에 맞는지 점검
- 개발되는 소프트웨어가 명세에 맞는지 점검

=> 요구사항대로 만들었는지를 검사하는거야

2. 확인 Validation

- 올바른 제품을 만들고 있는가?
- 소프트웨어가 고객의 기대에 맞는지 기대하는 것이다.
- 명세에 따르는 것을 넘어 고객이 기대하는 것을 제공하는지 확인
- 소프트웨어를 위해 비용을 지불하는 사람들이 기대하는 기능을 제공하는지를 점검

=> 요구사항을 고려하지 않은채 일단 사용자한테 도움이 되는건지 확인하는거야

=> 이 둘의 목표는 소프트웨어 시스템이 “목적에 맞는다”는 확신을 세우는 것이다.

이 확신의 수준은 “시스템의 목적, 시스템 사용자의 기대, 현재 시장 환경” 등에 좌우

5. 인스펙션 및 테스팅 (Inspection and testing)

〉 소프트웨어 테스팅과 검증과 확인 프로세스는 소프트웨어 인스펙션 및 리뷰를 포함

1. 인스펙션 (검사) , 리뷰 Inspection

- 요구사항, 설계, 소스코드를 분석하고 검사하여 문제를 찾음
- 프로그램을 실행시키지 않는 정적 V&V 기술
- 주로 소스코드를 대상으로 하지만 소프트웨어의 어떤 표현이라도 검사 가능

2. Testing

- 프로그램을 실행시켜 동작을 관찰하는 동적 V&V 기술

3. 인스펙션의 장점

- 테스팅 중에는 오류가 다른 오류를 가릴 수 있지만, 인스펙션은 정적 프로세스이므로 오류 간의 상호작용과 무관하여 한 번에 여러 오류를 발견할 수 있음
- 추가 비용 없이 시스템 불완전한 버전을 검사할 수 있고 불완전한 프로그램을 테스트 하려면 추가코드가 필요하다
- 인스펙션은 결함을 찾는 것뿐만 아니라 표준 준수, 이식성(portability) , 유지보수성 (maintainability) 등 폭넓은 품질 속성을 검토할 수 있음

  -〉 유지보수하고 변경하기 어렵게 만드는 비효율성, 부적절한 알고리즘 등 찾아낼수있다.

4. 인스펙션(정적)과 테스팅(동적)은 상호 보완적

- 인스펙션이 결함을 발견하는데 테스팅보다 효과적
- 인스펙션이 테스팅을 대체할 수 없음

인스펙션의 단점으로는 프로그램의 다른 부분 간의 예상치 못한 시스템 컴포넌트 간의 상호작용이나 타이밍 문제, 또는 시스템 성능 문제 때문에 발생하는 결함을 발견하는 데는 적합하지 않다. 작은 회사 또는 개발팀에서는 모든 잠재적인 팀 구성원이 소프트웨어 개발자가 될 수 있으므로 별도의 인스펙션 팀을 두는 것은 어렵고 비용이 많이 든다.

6. 테스팅

1. 테스트 케이스

- 무엇이 테스트되고 있는지에 대한 서술
- 테스트 입력 -> 예상 출력
    - 테스트 데이터 : 시스템을 테스트하기위한 입력 / 테스트 결과 : 테스트 데이버 입력에 따른 예상 출력

2. 상용 SW 시스템의 테스팅 3단계

1) 개발 테스팅 (development testing)

- 버그와 결함을 찾기 위해 개발 중에 테스팅
- 시스템 설계자와 프로그래머가 수행

2) 릴리스 테스팅 (release testing)

- 요구사항이 만족되는지 완성된 버전을 사용자에게 릴리스하기 전에 테스팅
- 목표 : 시스템이 이해당사자들의 요구사항에 맞는지 점검
- 별도의 테스팅 팀이 수행

3) 사용자 테스팅 (user testing)

- 시스템의 인수여부를 결정
- (잠재적) 사용자들이 자신의 환경에서 시스템을 테스팅
- 인수 테스팅 시스템 공급자로부터 시스템을 인수해야 하는지 아니면 추가 개발이 필요한지 결정하기 위해 고객이 공식적으로 시스템을 테스트하는 사용자 테스팅의 한 유형

3. 테스팅 프로세스

1. 수동 테스팅

- 테스터가 프로그램을 어떤 테스트 데이터를 가지고 실행시키고 그 결과를 예상되는 값과 비교한다.

테스터는 불일치를 기록하고 프로그램 개발자에게 보고한다.

2. 자동화된 테스팅 (Test automation)

- 테스트가 개발중인 시스템이 테스트될 때마다 실행되는 프로그램으로 포함되어 있다.
    - 수동 테스트보다 빠르다
    - 프로그램이 예정된 대로 동작하는지 점검만 가능하므로 테스팅이 완전히 자동화 될 수 없다.
    - 테스트 주도 개발 (TTD)을 위해서는 자동화된 테스팅이 필수
    - spec -> 테스트 케이스 -> 코딩 -> 테스팅

3. 회귀 테스팅(regression testing)

프로그램의 변경으로 새로운버그가 생기지 않았는지 점검하기위하여 이전에 수행한 테스트를 다시실행

8.1 개발 테스팅

: 시스템을 개발하는 팀에 의해 수행되는 모든 테스팅 활동을 포함

: 결함 테스트와 관련

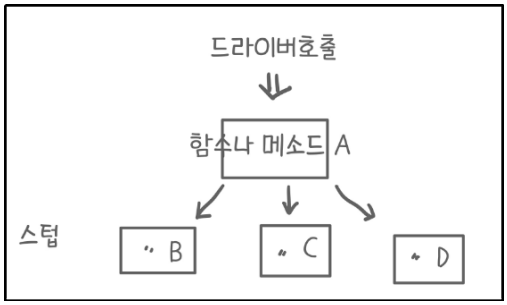
1. 개발 테스팅 3단계

*스텁 : A sutb is a short article in deed of expansion

*테스트 드라이버

1.시스템 및 시스템 컴포넌트를 시험하는 환경의 일부, 시험을 지원하는 목적 하에 생성된 코드

2.테스트 할 소프트웨어 또는 시스템을 제어하고 동작시키는데 사용되는 도구



1) 단위 테스팅 unit testing = 기능

- 함수, 클래스 등 프로그램의 단위를 테스팅
- 모듈의 기능, 단위의 기능을 테스트하는데 집중
- 테스트 드라이버 (test driver) / 스텁 (stub) 필요

2) 컴포넌트 테스팅 component testing = 인터페이스

- "통합 테스팅"이라고도 부른다.(integration testing)
- 전체가 아니라 일부만 붙여 테스트하는 거
- 여러 개별 단위들이 복합 컴포넌트를 생성하기 위해 통합된다.
- 컴포넌트의 인터페이스를 테스트하는데 집중
- 모듈간의 상호작용을 테스트 (모듈 통합 방법 : 빅뱅 테스트, 하향식 통합, 상향식 통합)

3) 시스템 테스팅 system testing = 통합

- 전체를 붙여서 테스트하는 거
- 모든 컴포넌트들을 통합하여 시스템의 기능을 전체로서 테스팅
- 요구사항이 만족되는지 테스트
- 컴포넌트의 상호작용을 테스트하는데 집중

2. 개발 테스팅의 목표

*디버깅 : 보통 코드의 문제를 찾아 그 문제를 고치기 위해 프로그램을 변경하는 프로세스

- 주로 버그를 발견하는 결함 테스트 - 디버깅과 중첩됨

8.1.1 단위 테스팅 Unit testing

1. 단위 테스팅

*모형객체 : 사용되는 외부 객체와 같은 인터페이스를 가지며 그 기능을 모사하는 객체

1. 개별적인 컴포넌트를 테스트

: 메서드 또는 객체 클래스와 같은 프로그램 컴포넌트를 테스팅하는 프로세스

2. 예 : 객체 클래스를 테스팅할 때

객체의 모든 기능이 포함되도록 테스트를 설계해야함

이는 객체에 포함된 모든 오퍼레이션들을 테스트해야 한다는 것을 의미

3. 일반적인 지침

- 클래스의 모든 오퍼레이션을 테스트

- 객체가 가질 수 있는 모든 상태를 거치도록 테스트 (모든 상태 전이 순서를 테스트)

- 상속된 오퍼레이션은 그것이 사용되는 모든 곳에서 테스트해야 한다.

(클래스 상속은 테스팅의 복잡도를 높인다.)

- 가능하다면, 언제든지 단위 테스팅을 자동화해야 함

> Junit 등의 테스트 자동화 프레임워크를 사용한다

(컴포넌트를 분리하여 테스트하기 위해 모형 객체 (스텁과 유사) 필요)

2. 자동화된 테스트 3부분

1. 설정 부분 : 시스템을 테스트 케이스로 입력과 예상되는 출력으로 초기화

2. 호출 부분 : 테스트될 객체나 메서드를 호출

3. 단언 부분 : 호출 결과를 예상되는 결과와 비교, 참으로 평가면 성공, 거짓이면 실패

8.1.2 단위 테스트 케이스 선정

: 비용과 시간이 많이 소요되므로 효과적인 단위 테스트 케이스 선정이 중요

1. 효과적인 단위 테스트

1. 컴포넌트가 예상(명세, specification)대로 동작하는 것을 보여야 함
2. 결함을 찾아야 함

2. 두 가지 종류의 테스트 케이스 설계

1. 프로그램의 정상적인 작업을 반영하고 컴포넌트가 동작한다는 것을 보여주어야 함
2. 공통적인 문제들이 발생하는 곳에 대한 테스팅 경험을 바탕으로 해야 함

3. 테스트 케이스 선정을 위한 효과적인 두 전략

1. 분할 테스팅

- 공통 특성을 가지고 동일한 방식으로 처리되어야 하는 입력 그룹을 식별한다.
- 각 그룹 안에서 테스트를 선정한다.
- 시스템과 컴포넌트 모두를 위하여 테스트 케이스를 설계하는 데 사용될 수 있다.
- 테케 설계에 대한 체계적인 접근법 하나는 시스템 또는 컴포넌트에 대한 입력과 출력 분할을 식별하는 것
    - 한번 분할의 집합을 식별하면 각 분할에서 테스트 케이스를 선정
    - 일반적으로 분할의 경계와 분할의 중간점에 가까운 케이스 선정

2. 가이드라인 기반 테스팅

- 어떤 종류의 테스트 케이스가 오류 발견에 효과적인지에 관한 지식을 포함한다.
- 프로그래머가 자주 범하는 오류의 종류에 대한 이전 경험을 반영해 테스트케이스를 선정

3. 코드 커버리지 > 이런 개념이 있다는 정도만 알기

- 소스 코드 중 테스트가 된 비율
- 문장(state)커버리지, 결정 또는 분기(decision or branch)커버리지, 조건 커버리지 등

3. 분할 테스팅 Partition Testing

1. 동등 분할 (equivalence partition) = 도메인

    - 입력 데이터 또는 출력 결과는 공통 특징을 가진 그룹으로 분할

    - 분할의 끝에 있는 입력을 처리할 때 프로그래머가 종종 범하는 오류를 처리하는 것을 도와줌

        - 양수 / 0 / 음수 , 자릿수 , 데이터 개수, 정상 / 오류

4. 블랙박스, 화이트박스 Black box & white box testing

1. 블랙박스 테스팅

    - 동등 분할을 식별하기 위하여 시스템의 명세를 사용할 때

    - 시스템이 어떻게 동작하는지 아무 지식도 필요하지 않음

    - 소프트웨어의 내부 구조나 작동 원리를 모르는 상태에서 동작을 검사하는 방식

    - 내부 구조 보지 않고 외부에서 이렇게 동작합니다란 스펙만 보고 테스팅하는 것

    - 테스트 케이스를 선정하기 위하여 시스템(테스팅 대상)의 명세만을 사용

    - 코드나 시스템 내부에 대한 지식을 사용하지 않고 테스트 케이스 선정

    - 예) 동등 분할

2. 화이트박스 테스팅

    - 다른 가능한 테스트를 찾기 위하여 프로그램의 코드를 보는 방식

    - 내부 소스 코드를 테스트하는 기법, 구현 기반 테스트

    - 구조 테스팅 (코드 - 내부 구조)

    - 알고리즘, 소스코드 등 시스템의 내부 구조를 고려하여 테스트 케이스 선정

    - 내부의 구조가 있으니 요리조리 테스트하는 것 - 코드 커버리지

    - 예) 문장 커버리지, 분기 커버리지, 단순 경로

=> 분할 테스팅은 블랙박스, 화이트박스 테스팅 둘 다 된다.

=> 코드 커버리지는 화이트박스 테스팅

5. Testing guidelines

1. 시퀀스(또는 배열 ,리스트 등) 프로그램 테스팅시 결함을 밝히는 데 도움 되는 지침

   - 원소가 하나인 시퀀스로 테스트

   - 테스트마다 서로 다른 크기를 가지는 시퀀스 사용

   - 시퀀스의 첫번째, 중간, 마지막 원소를 접근하도록 테스트 => 분할 경계 문제 확인

   - 길이가 0인 시퀀스 (원소가 없는)로 테스트

2. 일반적인 지침

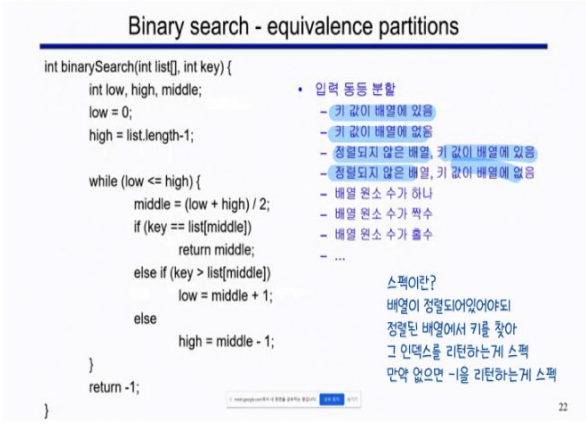
   - 모든 오류 메시지가 생성되도록 입력 값을 선택

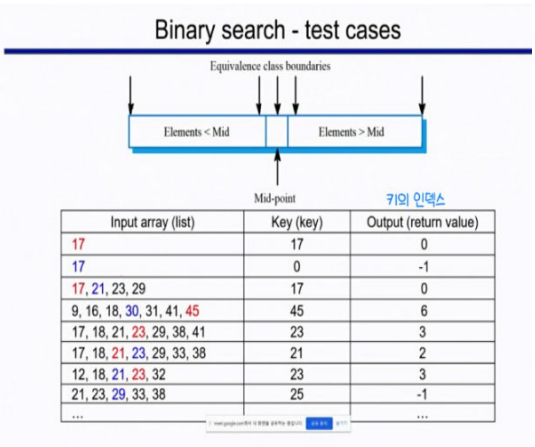
   - 버퍼 오버플로우가 일어나도록 입력을 선정

   - 같은 입력 또는 같은 입력 순서를 여러번 반복

   - 유효하지 않은 출력이 생성되도록 함

   - 계산 결과를 너무 크거나 너무 작게 되도록 함





* 경로 테스팅

- 컴포넌트나 프로그램의 모든 독립적인 실행 경로를 수행하는 것을 목표로 하는 테스팅 전략

- 모든독립적인경로가 실행된다면 컴포넌트의 모든문장들이 적어도 한번은 실행되었어야 한다.

- 모든 조건문은 참과 거짓인 경우 둘다 테스트되어야 한다.

- 객체와 연관된 메서드를 테스트하는 데 사용될 수 있다.

PDF) Flow graph

1. 흐름 그래프

- 프로그램 내의 경로를 보여줌

2. 순환 복잡도 cyclomatic complexity

1. 프로그램에 존재하는 독립적인 경로의 수

2. 경로의 조합은 고려하지 않음

3. 흐름 그래프 이용 계산

    - 순환 복잡도 = 간선수 - 노드수 + 2

4. 조건의 수 이용 계산

    - 순환복잡도 = 단순 조건의 수 + 1

3. 기본 경로 테스팅

- 독립적인 경로를 적어도 한 번은 거치도록 테스트 케이스를 선정

4. Junit과 EclEmma를 이용한 테스팅 실습

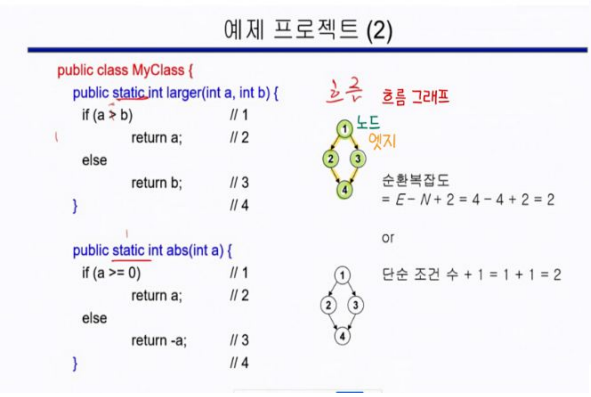
- if하고 줄바꾸고 else하고 줄 바꾸면서 코드 작성

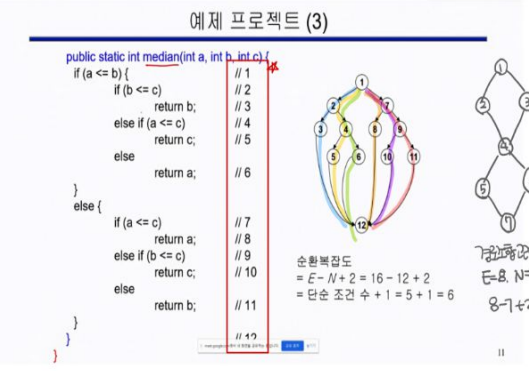
- 조건 뒤에는 반드시 줄 바꾸고 실행문이 있으면 번호를 붙힌다.

- 단, else에는 번호를 붙히지 않음

- 함수의 맨 끝라인에 번호를 붙힘

- 단순 조건은 &&나 ||이 없는 하나의 조건일 경우를 의미





```
static int binarySearch(int list[], int key) {
    int low, high, middle;          // 1
    low = 0;
    high = list.length-1;

    while (low <= high) {           // 2
        middle = (low + high) / 2;
        if (key == list[middle])
            return middle;
        else if (key > list[middle])
            low = middle + 1;
        else
            high = middle - 1;
    }
    return -1;
}                                   // 10
```

Source code에 번호를 직접 붙이고 flow graph를 완성해 봅시다. 분기가 일어나지 않는 것은 하나로 취급하면 10번 노드까지 필요합니다.

Cyclomatic Complexity =

```
void topologicalSort() {
1.      int top = 0, i, id;
        int inDegree[numOfTasks];
        int *stack[numOfTasks];
2.      for (id=0; id<numOfTasks; id++) {
3.          int numOfPred = task[id].getInDegree();
            inDegree[id] = numOfPred;
4.          if (numOfPred == 0) {
5.              stack[top] = id;
                top++;
            }
        } // end for
6.      for (i=0; i<numOfTasks; i++) {
7.          top--;
            id = stack[top];
            topOrder.push_back(id);
            Edge *e = task[id].getSucc();
8.          while (e) {
9.              int succId = e->getSuccTask();
                inDegree[succId]--;
10.             if (inDegree[succId] == 0) {
11.                 stack[top] = succId;
                    top++;
                }
12.             e = e->getSucc();
            } // end while
        } // end for
13. }
```

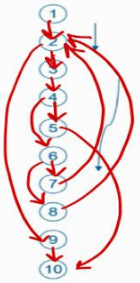
Flow graph(흐름 그래프)를 완성하고 cyclomatic complexity (순환복잡도)를 계산해 봅시다.

Cyclomatic Complexity =

```
static int binarySearch(int list[], int key) {
    int low, high, middle;          // 1
    low = 0;
    high = list.length-1;

    while (low <= high) {           // 2
        middle = (low + high) / 2;  // 3
        if (key == list[middle])    // 4
            return middle;          // 5
        else if (key > list[middle]) // 6
            low = middle + 1;       // 7
        else
            high = middle - 1;      // 8
    }
    return -1;                      // 9
}                                   // 10
```

Source code에 번호를 직접 붙이고 flow graph를 완성해 봅시다. 분기가 일어나지 않는 것은 하나로 취급하면 10번 노드까지 필요합니다.



Cyclomatic Complexity =

N = 10

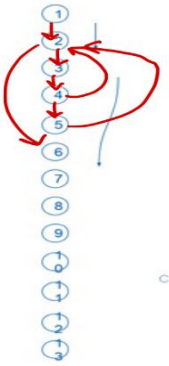
E = 12

12 − 10 + 2 = 4

3 + 1 = 4

```
void topologicalSort() {
1.      int top = 0, i, id;
        int inDegree[numOfTasks];
        int *stack[numOfTasks];
2.      for (id=0; id<numOfTasks; id++) {
3.          int numOfPred = task[id].getInDegree();
            inDegree[id] = numOfPred;
4.          if (numOfPred == 0) {
5.              stack[top] = id;
                top++;
            }
        } // end for
6.      for (i=0; i<numOfTasks; i++) {
7.          top--;
            id = stack[top];
            topOrder.push_back(id);
            Edge *e = task[id].getSucc();
8.          while (e) {
9.              int succId = e->getSuccTask();
                inDegree[succId]--;
10.             if (inDegree[succId] == 0) {
11.                 stack[top] = succId;
                    top++;
                }
12.             e = e->getSucc();
            } // end while
        } // end for
13. }
```

Flow graph(흐름 그래프)를 완성하고 cyclomatic complexity (순환복잡도)를 계산해 봅시다.



Cyclomatic Complexity =

5 + 1 = 6

17 − 13 + 2 = 6

8.1.3 컴포넌트 테스팅 Component testing

1. 컴포넌트 테스팅

〉 컴포넌트 테스팅의 정의 대신 “컴포넌트의 인터페이스를 테스트”한다 정도로만 알기

- 인터페이스 테스팅라고도 부른다

- (복합) 소프트웨어 컴포넌트는 대개 상호 작용하는 여러 객체들로 이루어져 있다.

- 컴포넌트의 테스팅은 컴포넌트의 인터페이스를 테스트하는데 중점

2. 인터페이스 유형 필요없을듯

- 매개변수 인터페이스

- 공유 메모리 인터페이스

- 프로시저 인터페이스

- 메시지 전달 인터페이스

3. 인터페이스 오류 3가지

- 복잡한 시스템에서 가장 흔한 형태의 오류 중 하나

1. 인터페이스 오용 misuses
    - 호출 사용법 관련 오류
    - 예 : 매개변수 유형 오류, 매개변수 순서 오류 등
2. 인터페이스 오해 misunderstanding
    - 인터페이스 명세를 잘못 이해
    - 예 : 이진 탐색 함수에 정렬되지 않은 배열을 전달
3. 타이밍 오류 timing
    - 공유 메모리나 메시지 전달 인터페이스를 사용하는 경우 일어남
    - 예 : 생산자/ 소비자 속도 또는 타이밍 오류

4. 인터페이스 테스팅을 위한 일반적인 지침

1. 외부 컴포넌트에 전달되는 매개변수 값이 범위의 극단에 있게 테스트 집합을 설계한다.

2. 포인터나 레퍼랜스를 사용할 경우 널 객체가 전달되는 경우를 테스트

3. 고의적으로 컴포넌트가 실패하도록 테스트를 설계

4. 메시지 전달 시스템에서 스트레스 테스팅을 사용

    - 타이밍 문제를 발견하는 효과적인 방법

5. 공유 메모리를 이용하는 경우 컴포넌트들이 활성화되는 순서가 바뀌도록 테스트를 설계

=> 때때로, 인터페이스 오류를 찾기위해 테스팅보다 인스펙션나 리뷰를 사용하는게 더 효과적

8.1.4 시스템 테스팅 System testing

1. 시스템 테스팅

- 시스템의 버전을 생성하기 위하여 컴포넌트들을 통합하고 통합된 시스템을 테스팅하는 것과 관련
- 컴포넌트들이 호환되는지, 올바르게 상호작용하는지, 인터페이스를 통하여 정확한 시간에 정확한 데이터를 전송하는지 검사한다.
- 시스템 구성요소들의 상호작용을 테스팅하는데 중점
- 재사용가능한 컴포넌트나 시스템을 새로운 컴포넌트와 통합시 예상대로 작동하는지 점검
    - 시스템의 창발적(emergent)특성을 테스트
    - 계획된 창발적 행동 / 예기치 못한 창발적 행동
- 유스케이스 기반 테스팅이 효과적
    - 요구사항에서 작성한 유스케이스 시나리오 -> 시스템 테스팅에 이용
    - 유스케이스 시나리오 표현 : 시퀀스 다이어그램 / 액티비티 다이어그램 / 자연어 등
    - 기본(basic) 흐름, 대안(alternatvie) 흐름, 예외(exceptional) 흐름을 고려
- 컴포넌트 테스팅과 겹치지만 차이점이 존재

2. 컴포넌트 테스팅과의 차이점

1. 시스템 테스팅 중에는 별도로 개발된 재사용 가능한 컴포넌트와 기성품 시스템이 새로 개발된 컴포넌트들과 통합될 수 있다. -> 완전한 시스템이 테스트됨
2. 다른 팀이나 하부팀에서 개발된 컴포넌트가 이 단계에서 통합되어 시스템 테스팅은 개별적이라기보단 집단적인 프로세스이다.

8.2 테스트 주도 개발

1. 테스트 주도 개발 Test-Driven Development (TDD)

- TTD는 테스팅과 코드 개발이 중첩시키는 프로그램 개발 접근법이다.
- 점증적인 접근법으로 증분(증가분 increment)의 테스트를 작성하고 코드를 개발
- 개발된 코드가 그것을 위한 테스트를 모두 통과할 때까지 다음 증분 작업을 시작하지 않음
- Junit 환경과 같이 자동화된 테스팅 환경에 필수적이다.
- 아주 작은 증분의 코드가 개발되기에 기능을 추가하거나 프로그램을 리팩토링했을 때마다 모든 테스트를 실행할 수 있어야 함
- 프로그래머가 코드의일부가 실제로 무엇을 하는 것인지에 대한 생각을 분명히하도록 도와줌
- 애자일 기법인 XP에서 시작됨

  > 테스트 우선 개발 Test First development

2. TDD 프로세스 단계

1. 증분을 위한 기능 식별
2. 테스트를 작성하여 자동화된 테스트(Junit)로 구현
   - 통과되었는지, 실패되었는지 여부를 보고할 것
3. 이전의 모든 테스트와 함께 추가한 테스트를 실행
   - 처음에는 기능 구현하지 않았으므로 새로운 테스트는 실패
   - 이는 테스트 집합에 그 테스트가 무엇을 추가했다는 것을 보여주도록 고의적인 것
4. 증분의 기능을 구현하고 테스트를 다시 실행
5. 모든 테스트를 통과하면 다음 증분을 개발

예) A/B - A 테스트케이스 만듬 - A시험(구현없으니 당연히 실패 뜸) - A구현 - A시험

- B도 마찬가지로 진행됨

3. TDD 장점

1. 코드 커버리지 code coverage : 모든 코드와 연관된 테스트가 있음
    - 코드 커버리지 100퍼 : 내가 만든 코드가 다 실행이 되어봤다는 것
2. 회귀 테스팅 regression testing : 모든 테스트를 실행
    - 변경이 새로운 버그를 초래하지 않았는지 회귀 테스트를 항상 실행할 수 있다.
    - 지난번에 성공했던 테스트를 다시 실행하는 것
3. 단순화된 디버깅 : 문제를 지역화하기 쉬움
4. 시스템 문서화 : 테스트를 보면 코드를 이해하기 쉬움 -> 테스트가 주석 역할을 해준다.

4. 회귀 테스팅

- 시스템을 변경한 후 변경과 관련된 테스트뿐만 아니라 기존에 성공한 모든 테스트를 실행
- 변경으로 인하여 새로운 버그가 생기지 않았고 새로운 코드가 기존 코드와 문제없이 동작하는지 검사
- Junit 등 자동화된 테스팅 필요
- TDD의 가장 중요한 장점 중 한개는 회귀 테스팅 비용을 감소시키는 것이다.
- 회귀 테스팅은 시스템의 변경이 가해진 후 성공적으로 테스트된 집합들을 실행시키는 것
- 회귀테스팅은 비용이 많이 든다 -> 돌려볼게 많아서 수작업으로 하기 어려움 -> 비효율적

=> 자동화된 테스팅으로 회귀테스팅의 비용을 극적으로 감소시킨다.

8.3 릴리스 테스팅 Release testing

1. 릴리스 테스팅

- 시스템 테스팅의 한 형태
- 외부에서 사용하기로 의도된 시스템의 특정 릴리스를 테스팅 하는 프로세스
- 보통 시스템 릴리스는 고객와 사용자를 위한 것

2. 릴리스 테스팅과 시스템 테스팅의 구분

- 개발팀이 아닌 별도의 팀이 테스팅 해야함
- 시스템이 요구사항을 만족시키는 것을 점검
- 시스템이 사용하기에 충분하다는 것을 확인 점검 프로세스
- 시스템 테스팅은 결함 테스팅에 중점

3. 블랙박스 테스팅 프로세스

- 소위 기능적 테스팅
- 테스트 케이스를 시스템 명세를 이용하여 작성
- 시스템의 기능만 고려하고 구현은 고려하지 않음

8.3.1 요구사항 기반 테스팅

- 각 요구사항을 고려하고 그것을 위한 테스트 집합을 유도하는 테스트 케이스 설계에 대한 체계적인 접근
- 검증 테스팅
- 추적 가능성 기록 관리 -> 테스트된 특정 요구사항과 테스트를 연결해준다.

8.3.2 시나리오 테스팅

- 시스템을 사용하는 전형적인 시나리오를 만들어 이 시나리오들을 시스템의 테스트 케이스를 개발하는 데 사용하는 릴리스 테스팅 접근법
- 유스케이스 시나리오
- 한 시나리오 내에서 여러 요구사항을 테스팅 -> 요구사항의 조합 점검

8.3.3 성능 테스팅 Performance testing

1. 성능 테스팅

*운영 프로파일 : 시스템이 처리할 작업의 실제 구성이 반영된 테스트들의 집합

- 일단 시스템이 완전히 통합되면 성능과 신뢰성과 같은 창발성에 대한 테스트가 가능해짐
- 시스템이 요구사항을 만족하는지 보여주는 것과 결함을 발견하는 것 둘다 관련되 있음
- 시스템의 운영 프로파일을 반영하여 트랙잭션 유형의 비율에 따라 테스트
- 시간 당 처리량, 응답시간 등을 측정

2. 스트레스 테스팅 stress testing

- 시스템의 최대 설계 부하를 초과하는 요청을 생성하여 한계에 가까운 테스트를 설계하는 것
- 스트레스 테스팅의 목적
    - 시스템의 장애 행동을 관찰 > 시스템이 망가지지 않고 부드럽게 실패하는지 점검

        부드럽게 실패 : 보통 장애가 생겨서 시스템이 죽거나 멈춰버리는 경우가 발생하는데 지금까지 했던게 싹 다 날아가는게 아니라 어느정도 반영을 하고 살릴거 살리고 데이터 손상이 적게 멈춰버리는 것

    - 시스템의 최대부하가 걸렸을 때만 보일 수 있는 결함 찾기

8.4 사용자 테스팅

1. 사용자 테스팅 User testing

- 사용자 또는 고객 테스팅은 입력을 제공하고 시스템 테스팅에 관한 의견을 내는 테스팅 프로세스의 한 단계
- 외부 공급자로부터 위탁받은 시스템을 공식적을 테스트하는 것을 포함할 수 있다.
- 광범위한 시스템 테스팅과 릴리스 테스팅이 수행된 경우에도 사용자 테스팅은 필수적
- 사용자의 작업 환경으로부터 영향은 시스템의 신뢰성, 성능, 사용성, 견고성 등에 영향

2. 사용자 테스팅의 유형

1. 알파 테스팅 alpha testing
    - 사용자가 개발자와 함께 개발자 사이트에서 테스트
    - 소프트웨어의 초기 릴리스를 테스트
    - 개발팀에게 명백하지 않은 문제점과 논점들을 사용자가 확인할 수 있음
    - 사용자들이 더 현실적인 테스트의 설계를 도와주는 실질적인 정보를 제공할 수 있다.
        - 소프트웨어 제품이나 앱을 개발시 자주 사용
        - 사용자들은 정보를 일찍 얻기 위해 알파 테스팅 프로세스에 관여하기 원할 수 있다.
        - 예상치 못한 변경이 비즈니스에 혼란스러운 영향을 줄 수 있는 리스트 감소
        - 맞춤 소프트웨어 개발시 사용될 수 있다.
2. 베타 테스팅 beta testing
    - 시스템의 초기 또는 가끔 완성되지 않은 릴리스를 평가 목적으로 사용
    - 고객 또는 큰 규모의 사용자 그룹이 시스템을 사용해보고 문제점 피드백
    - 다양한 설정과 운영 환경에서 사용되는 시스템을 테스트하는데 효과적
    - 소프트웨어와 운영 환경의 특성 간 상호 작용 문제를 발견하기 위해 사용된다.
3. 인수 테스팅 acceptance testing
    - 시스템의 인수를 결정하기 위해 고객의 환경에서 고객의 데이터를 이용
    - XP에서는 별도의 인수 테스팅이 없을 수도 있음

9장 소프트웨어 진화

1. 소프트웨어 변경은 불가피함

- 소프트웨어를 사용하면서 새로운 요구사항이 생김
- 변화하는 비즈니스 환경에 적응
- 운영 중 발견된 오류를 수정
- 하드웨어와 소프트웨어 플랫폼이 변화
- 성능이나 신뢰성 등 비기능성 특성 개선

2. 소프트웨어 진화의 중요성

- 조직의 소프트웨어는 중요한 비즈니스 자산
- 자산의 가치를 유지하기 위해선 변경이 필요함
- 대규모 기업은 기존 시스템 유지보수에 더 많은 비용을 지출

  (소프트웨어 비용의 60% 이상이 진화 비용)

9.1 진화 프로세스

1. 진화 프로세스에 영향을 주는 것

1. 유지 보수되는 소프트웨어의 유형
2. 조직이 사용하는 소프트웨어 개발 프로세스
3. 참여한 사람들의 기술이나 경험에 좌우

2. 변경 제안이 시스템 진화를 주도

- 변경 제안 = 변경 요청
- 모든 조직에서 정형적 또는 비정형적 시스템 변경 제안이 시스템 진화를 주도
- 릴리스된 시스템에 구현되어 있지 않은 기존 요구사항, 새로운 요구사항에 대한 요청,시스템 이해당사자로부터의 버그 보고, 시스템 개발팀이 제안하는 소프트웨어 개선을 위한 새로운 아이디어를 기반으로 할 수 있다.
- 변경 제안이 받아들여지기 전에, 어떤 컴포넌트들이 변경되어야 하는지 알아내기 위한 소프트웨어 분석이 필요
  > 이 분석은 변경의 비용 및 영향의 평가를 가능하게 함
- 변경 식별 및 시스템 진화 프로세스는 시스템의 수명 동안 계속된다.

3. 변경의 비용 및 영향이 평가됨

1. 개발과 진화가 통합된 상황

- 변경 구현은 단순히 개발 프로세스의 반복이다

2. 개발과 진화가 다른 팀에 의해 수행되는 상황

- 개발과 차이점 : 변경 구현의 초기단계에서 프로그램의 이해하는 단계가 필요
- 새로운 개발자들이 프로그램의 구조와 제안된 변경이 프로그램에 미치는 영향을 이해

3. 요구사항 명세와 설계 문서를 수정해야 하는 경우

- 새로운 소프트웨어 요구사항들을 기록, 분석, 확인해야 함
- UML 모델 등 설계 문서를 수정해야 함

4. 변경 요구가 긴급하게 해결되어야 하는 상황

- 신속한 변경 필요성은 모든 SW문서를 업데이트할 수 없다는 것을 의미

> 요구사항과 설계를 수정하지 않고 프로그램을 긴급하게 수정

> 따라서 요구사항, 설계, 코드가 일치하지 않게 되는 결과 초래

4. 변경 요구가 긴급하게 변경해야 하는 3가지 이유

1. 심각한 시스템 결함이 감지될때

2. 시스템 운영 환경의 변화가 정상적인 운영을 중단시키는 예상치 못한 영향을 미칠때

3. 비즈니스 운영에 대한 예측하지 못한 변화가 있을 때

5. 애자일 방법과 진화 Agile methods and evolution

1. 애자일 방법은 개발뿐만 아니라 진화를 위해 사용 가능하다.

> 애자일 방법은 점증적인 개발이므로 개발과 진화가 매끄럽게 연결됨

2. 개발팀에서 시스템 진화를 책임지는 별도의 팀으로 일 넘기는 과정에서 문제 발생

1) 개발팀은 애자일 접근법 / 진화팀은 계획 기반 접근법을 선호했을 경우

- 진화팀은 진화를 지원하기 위한 자세한 문서 예상가능
- 개발팀은 애자일이므로 그러한 문서를 거의 생성하지 않음

=> 시스템변경에 따라 수정될수있는 시스템 요구사항에 대한 확실한서술이 없을수도 있다.

2) 개발팀은 계획 기반 접근법 / 진화팀은 애자일 방법을 선호했을 경우

- 진화팀은 자동화된 테스트 개발을 처음부터 다시 시작해야 할 수 있다.
- 애자일 개발에서 예상되는 리팩토링과 단순화가 되지 않은 상태일 수도 있음

=> 애자일 개발 프로세스에 사용하기 전에 코드를 개선하는 프로그램 재공학이 요구

3. 테스트 주도 개발과 자동화된 회귀 테스팅은 시스템 변경이 있을 때 유용

- 시스템 변경은 사용자 스토리로 표현할 수 있음
- 고객의 참여는 변경 사항의 우선 순위를 정하는데 도움이 될 수 있음

4. 수행되어야 할 작업 목록에 초점을 맞추는 스크럼 접근법은 변경 사항의 우선 순위를 정하는데 도움이 됨

5. 진화는 잦은시스템 릴리스를 기반으로하는 개발프로세스의 연속으로 생각될수 있음

9.2 레거시 시스템 Legacy systems

1. 레거시 시스템 > 레거시 시스템은 용어만 알면 된다라고 하신 것 같음

- 구형 소프트웨어 시스템들을 의미
- 신형 시스템 개발에는 더 이상 사용되지 않는 언어와 기술에 의존하는 구형 시스템
- 보통 오랜 시간 동안 유지보수되어왔고 변경들로 인해 그 구조가 저하되어 있다.
- 레거시 SW는 새로운 프로세스를 지원하기 위해 수정될 수 없기 때문에 보다 효과적인 비즈니스 프로세스를 위한 변경이 불가능할 수도 있다.
- 새로 개발하면 돈도 많이 들고 이때까지 잘 되는데 굳이 새로 개발하면 초기에는 문제가 많이 발생하니까 구형 시스템을 고집하는 것
- 레거시 사용하는 부문 예시 : 금융, 은행, 증권 같은 곳

2. 레거시 구성 요소 > 그렇게 중요해 보이지 않음

1. 시스템 하드웨어
2. 지원 소프트웨어
    - OS, 유틸리티, 컴파일러 등
3. 애플리케이션 소프트웨어
    - 다른 시기에 개발된 여러 개의 프로그램으로 구성
4. 애플리케이션 데이터
    - 막대한 양의 데이터가 시스템 수명동안 쌓여 왔음
    - 이 데이터는 일관성이 없고 여러 파일에 중복, 여러 데이터베이스에 걸쳐있음
5. 비즈니스 프로세스
    - 레거시 시스템을 기반으로 설계된다.
    - 레거시 시스템의 기능에 의해 제약을 받음
6. 비즈니스 정책과 규칙

3. 레거시 시스템의 문제점

1. 기술(자) 부족
   - COBOL은 비즈니스 시스템을 작성하기 위해 설계된 프로그래밍 옛 언어
   - 레거시 코드가 COBOL언어로 작성 되어 있음
   - 이러한 프로그램은 여전히 효과적이고 효율적으로 작동하고 있으며,
     이를 이용하고 있는 기업들은 프로그램을 변경할 필요를 느끼지 않음
   - 하지만, 시스템의 원래 개발자들이 은퇴함에 따라 COBOL 프로그래머가 부족
2. 보안 취약점 존재
   - 레거시 시스템이 인터넷이 광범위하게 사용되기 전에 개발되어 보안 취약점 포함
3. 인터페이스 문제
   - 현대적인 프로그래밍 언어로 작성된 시스템과 인터페이스하는 데 문제 발생
4. 하드웨어 유지보수 문제
   - 원래의 SW도구 공급 업체는 사업을 그만두었거나 시스템 개발하는 데 사용되는
     지원 도구를 더 이상 유지보수하지 않을 수 있다.
   - 시스템 하드웨어는 더 이상 쓸모가 없고 유지보수 하는 데 점점 더 비용이 증가

4. 레거시 시스템 교체하지 않는 이유

- 교체하기엔 비용이 너무 많이 들고 교체하는 리스크가 너무 크기 때문에
- 만약 레거시 시스템이 효과적으로 동작시 교체 비용이 새 시스템 도입으로 지원
  비용을 초과하니까 교체하지 않음

5. 레거시 시스템을 교체시 비싸고 위험한 이유

1. 레거시 시스템은 완전한 명세서가 거의 없다
   - 원래의 명세서가 사라졌을 수도 있다.
   - 존재하더라도 모든 변경 사항이 업데이트되어 있을 가능성 낮다
   - 따라서, 사용 중인 레거시 시스템과 기능적으로 동일하도록 새로운 시스템을 명세화하는 간단한 방법은 없다.
2. 비즈니스 프로세스와 레거시 시스템이 서로 밀접하게 얽혀 동작한다.
   - 시스템이 교체되는 경우에 비즈니스 프로세스가 예상치 못한 비용과 결과를 감수하고 변경되어야 한다.
3. 중요한 비즈니스 규칙이 소프트웨어 안에만 내장되어 있고 문서화 되어 있지 않음
   - 비즈니스 규칙을 위반하는 것은 예측할 수 없는 결과를 초래 할 수 있다.
4. 새로운 시스템개발은 리스크가 높고 새로운시스템에 예기치않은문제가 있을수있다.
   - 새로운 시스템은 제때 예상된 가격에 제공되지 않을 수 있다.

6. 레거시 시스템이 변경하기 비싼 이유

1. 다양한 사람들이 시스템 변경했기에 프로그램 스타일 및 사용 규칙이 일관성 없다
   - 코드 이해하는데 어려움
2. 시스템 부분, 전체가 구식 프로그래밍 언어를 사용하여 구현되어 있어 이에 대한 지식을 가진 사람을 찾기 어려워서 비싼 아웃소싱으로 시스템 유지보수 해야함
3. 시스템 문서는 종종 불충분하고 뒤떨어있다.
   - 어떤 경우엔 소스 코드가 유일한 문서일 수 있다.
4. 다년간 유지보수는 보통 시스템 구조를 저하시키고 점점 이해하기 어려움
5. 시스템은 느린 구형 하드웨어에서 효과적으로 동작하도록 공간 활용 또는 실행 속도에 최적화되었을 수 있다.
6. 데이터가 호환되지 않는 구조를 가진 다양한 파일들로 관리 될 수 있다.

=> 사용 중인 레거시 시스템을 유지하면 교체의 리스크를 피할 수 있으나, 시스템이 오래 될수록 기존 소프트웨어를 변경하는 것이 필연적으로 더 비싸진다.

9.2.1 레거시 시스템 관리

1. 레거시 시스템 진화시키기 위한 전략적인 4가지 옵션

1. 시스템의 완전한 폐기
   - 비즈니스 프로세스에 효과적으로 기여하지 못할 때 선택
   - 비즈니스 프로세스가 변경되어 더이상 레거시 시스템에 의존하지 않을 때 사용
2. 정기적인 유지보수를 계속
   - 시스템이 필요하고 안정적이지만 사용자가 시스템 변경 요청이 거의 없을 경우
3. 유지보수성을 향상시키기 위해 시스템 재공학
   - 변경으로 인해 시스템 품질이 나빠졌고, 시스템에 대한 변경이 여전히 요구되는 경우
4. 시스템 전체나 일부를 새로운 시스템으로 대체
   - 새로운 하드웨어와 같은 요인으로 구형 시스템을 계속 운영할 수 없을 때나
     기성품 시스템을 이용하여 합리적인 가격으로 새로운 시스템을 개발할 수 있을 때 선택

2. 품질, 비즈니스 가치에 대한 네가지 부류의 시스템 271-272

1. 낮은 품질, 낮은 비즈니스 가치
2. 낮은 품질, 높은 비즈니스 가치
3. 높은 품질, 낮은 비즈니스 가치
4. 높은 품질, 높은 비즈니스 가치

9.3 소프트웨어 유지보수

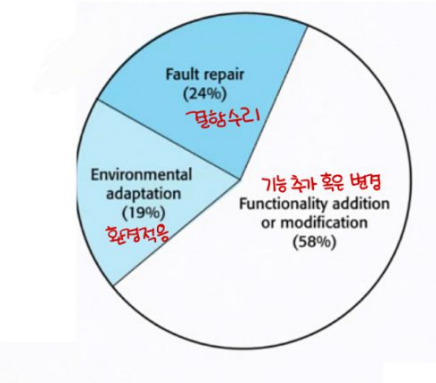
1. 3가지 유형의 소프트웨어 유지보수 275-276

1. 버그와 취약점을 고치기 위한 결함 수리

2. 소프트웨어를 새로운 플랫폼과 환경에 맞추기 위한 환경적 적응

3. 새로운 기능을 추가하고 새로운 요구사항을 지원하기 위한 기능성 추가

2. 그림 9.12에 대한 설명



- 1980년에서 2005년까지의 기존 연구들의 유지보수 비용 분포를 비교한 것

- 저자들은 30년 동안 유지보수 비용의 분포가 거의 변화하지 않았다는 것을 발견

3. 초기 개발때 새로운 기능 추가보다 유지보수때 추가가 비용 많이 드는 이유

1.새로운 팀이 유지보수되는 프로그램을 이해해야 함

2. 유지보수와 개발의 분리는 개발 팀이 유지보수하기 쉬운 소프트웨어를 작성하는데

   인센티브가 없다는 것을 의미 = 즉 이득 없다

3. 프로그램 유지보수 업무는 인기가 없음

4. 프로그램이 오래될수록 구조가 저하되고 변경하기 어려워짐

   - 결과적으로 이해하고 변경하기 어려워짐

4. 문제점 원인과 해결방안

- 원인

  : 아직도 많은 조직들이 소프트웨어 개발과 유지보수를 별개의 활동으로 간주하기에

- 해결 방안

  1. 시스템이 개발 프로세스에서 계속 진화해 나간다고 생각하기
  2. 유지보수는 새로 소프트웨어를 개발하는 것만큼 높은 중요도 갖기
  3. 반복되는 수정에 의해 구조 망가짐
     1) 소프트웨어 재공학 기술이 시스템 구조와 이해도를 향상시키기 위해 적용
     2) 리팩토링이 시스템 코드의 품질을 개선하고 변경을 용이하게 만들 수 있음
  4. 미래의 변경 비용을 줄이기 위해 시스템 설계하고 구현하는데 노력 많이 투자
  5. 개발 중에 소프트웨어를 구조화하고 이해하고 변경하기 쉽게 만들어 주기
  6. 좋은 소프트웨어 공학 기술은 모두 유지보수 비용을 절감할때 도움이 됨

9.3.1 유지보수 예측

: 소프트웨어 시스템에 요구될 수 있는 변경을 평가하는 것과 변경하는데 가장 비용이 많이 들 것 같은 시스템의 부분들을 식별하는 것에 관한 것

1. 유지보수성을 평가할 수 있는 지표

1. 수정 유지보수를 위한 요청 횟수
   - 버그 및 결함 보고 횟수의 증가는 유지보수 프로세스 동안 고쳐진 오류보다 더 많은 오류들이 프로그램에 생겼음을 나타낼 수 있다. -> 유지보수성의 하락을 의미
2. 영향 분석에 필요한 평균 시간
   - 영향 분석에 요구되는 시간이 증가한다는 건 더 많은 컴포넌트들이 영향을 받고 유지보수성이 감소한다는 것을 의미
3. 변경 요청를 구현하는데 걸리는 평균 시간
   - 변경을 구현하는데 필요한 시간의 증가는 유지보수성의 하락을 나타낼 수 있다.
4. 미해결된 변경 요청 개수
   - 이 개수가 시간에 따라 증가하면, 유지보수성이 하락하는 것을 의미

9.3.2 소프트웨어 재공학 Software reenginerring

1. 소프트웨어 재공학

- 재공학이란 , 시스템의 전체 또는 일부를 재구조화하거나 다시 작성하는 것
- 재공학은 소프트웨어를 이해하고 변경하기 쉽게 재구조하고 재문서화하는 것이다.
- SW 유지보수는 변경되어야 하는 프로그램을 이해하고 요구된 변경을 구현하는 것을 포함
- 레거시 SW 시스템을 유지보수하기 쉽게 만들기 위해, 그 구조와 이해용이성을 개선하기 위해 시스템을 재공학 할 수 있다.
- 시스템을 다시 문서화하고, 시스템 아키텍처를 리팩토링하고, 프로그램을 최신의 프로그래밍 언어로 변환하고, 시스템 데이터의 구조와 값을 수정하고 업데이트하는 것을 포함한다.
- 소프트웨어의 기능은 변경되지 않으며, 보통 시스템 아키텍처에 대한 중대한 변경을 피하도록 노력해야한다.

2. 재공학 교체에 비해 중요한 두가지 장점

1. 리스크 감소
   - 다시 개발하는 것에는 매우 높은 리스크가 존재 -> 교체다 다시개발보단 리스트 적음
2. 비용 절감
   - 새로운 소프트웨어를 개발하는 비용에 비해 재공학 비용은 상당히 적을 수 있다.

3. 재공학 프로세스의 활동

1. 소스 코드 변환 : 오랜 프로그래밍 언어를 최신 버전이나 다른 언어로 변환
2. 역공학 reverse engineering : 프로그램을 분석하고 정보를 추출한다.
3. 프로그램 구조 개선
   - 프로그램 제어 구조가 분석되어 읽고 이해하기 쉽게 수정된다.
4. 프로그램 모듈화
   - 프로그램의 관련된 부분들이 함께 그룹으로 묶이고, 적절한 경우에 중복성이 제거
   - 또는 아키텍쳐 변환을 포함할 수 있다.
5. 데이터 재공학
   - 데이터베이스 스키마 재정의
   - 기존 데이터베이스를 새로운 구조로 변환하는 것도 포함
   - 일반적으로 데이터도 정리
   - -> 매우 비용이 많이 들고 장기적인 프로세스이다.

9.3.3 리팩토링 Refactoring

1. 리팩토링 = 소스코드 개선 , 예방 유지보수

- 변경에 따른 프로그램의 품질 저하를 늦추기 위하여 프로그램을 개선하는 것
- 구조를 개선하고 복잡성을 줄이고 보다 이해하기 쉽도록 프로그램을 수정하는 것을 의미
- 리팩토링에서는 프로그램의 기능을 추가하지 않으며 개선에 집중함
- 프로그램의 장기 유지보수 비용을 줄이는 효과적인 방법이다.
- 기능을 보존하면서 프로그램의 작은 변경을 가하는 것으로 예방 유지보수라고도 함

2. 애자일 개발과 리팩토링

- 애자일 개발자는 이러한 품질 저하를 방지하기 위해 잦은 리팩토링 필요
- 회귀 테스트에 대한 강조는 리팩토링을 통해 새로운 오류를 생겨나는 리스크를 낮춘다.

3. 재공학과 리팩토링

1) 공통점

: 모두 소프트웨어를 이해하고 변경하기 쉽게 하기 위한 것

2) 차이점

- 재공학은 시스템이 일정 시간 동안 유지보수 후 유지 보수 비용이 증가하고 있을 때 발생
- 재공학은 유지보수성이 더 좋은 새로운 시스템으로 만들기 위해 레거시 시스템을 처리하고 재공학하는 데 자동화된 도구를 사용
- 리팩토링은 개발과 진화 과정 전반에 걸쳐 개선하는 연속 프로세스이다.
- 리팩토링은 시스템 유지보수 비용과 어려움을 증가시키는 구조 및 코드의 품질 저하를 방지
- 리팩토링이 더 작은 단위이고 소스코드 개선, 기능 추가 없음

4. 리팩토링으로 개선할 수 있는 예

> 외우라는게 아니라 리팩토링은 소스코드 개선이라고 말하기 위함

1. 중복 코드

   - 동일하거나 유사한 코드가 여러 곳에 있는 경우 하나의 메서드나 함수로 구현하는 호출

2. 길이가 긴 메소드

   - 길이가 너무 긴 메서드는 짧은 메서드들로 재설계

3. 스위치 케이스문

   - 다형성으로 대체할 수 있는 경우

4. 데이터 군집

   - 동일한 그룹의 데이터 항목이 여러곳에서 나타날때 데이터를 캡슐화하는 객체로 대체

5. 추측에 근거한 불확실한 일반성

   - 앞으로 필요할 때를 대비하여 프로그램에 일반성을 포함시킨 경우 간단하게 제거

10장 확실성 있는 시스템

- SW 집약적 시스템은 정부, 기업, 개인에게 매우 중요하기에 신뢰할 수 있어야한다.

- 즉, 소프트웨어 시스템을 신뢰할 수 있어야 한다.

1. 확실성 dependability

- 가용성, 신뢰성, 안전성, 보안성과 같은 관련된 시스템 특성들을 포함하기 위해 제안

2. 확실성이 중요한 이유

1. 시스템 장애가 많은 사람들에게 영향

    - 시스템의 가용성에 영향을 주는 시스템 장애는 잠재적으로 모든 사용자에게 영향 줌

    - 사용할 수 없는 시스템은 정상적인 비즈니스가 불가능하다는 것을 의미

2. 신뢰할 수 없고 안전하지 않고 보안성이 없는 시스템은 사용자가 거부한다.

3. 막대한 시스템 장애 비용이 발생한다.

4. 확실성이 없는 시스템은 정보 손실을 초래할 수 있다.

3. 중대한 시스템 critical system

- 시스템 장애가 사람이나 환경에 피해를 주거나 큰 경제적 손실을 줄 수 있는 시스템의 종류

- 예시) 안전성 중심 시스템, 임무 중심 시스템, 비즈니스 중심 시스템

  1) 안정성 중심 : 인슐린 펌프와 같은 의료 기기에 들어가는 임베디드 시스템

  2) 임무 중심 : 우주선의 네비게이션 시스템

  3) 비즈니스 중심 : 온라인 송금 시스템

4. 확실성이 높지 않아도 유용한 시스템

- 필자는 시스템 장애로 인해 발생할 수 있는 피해를 최소화하는 조치를 취함으로써 시스템

  확실성의 부족을 보완한다.

5. 확실성 있는 시스템 설계시 고려 (장애의 원인) (causes of failure)

1. 하드웨어 장애

    - 주로 부품고장, 물리적인 환경요인, 부속품의 수명이 다했기 때문에 발생

2. 소프트웨어 장애

    - 명세화 오류, 설계 오류, 구현 시의 실수 등으로 발생

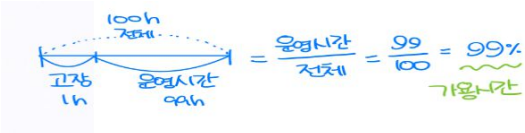
3. 운영 시의 장애

    - 의도하지 않는 방법으로 사용했거나, 운영하지 않았기에 발생

10.1 확실성의 특성들

1. 확실성의 5가지 주요 측면 Principal properties + 영어도

1. 가용성 (availability)



- 딱 어떤 시점을 찍었을 때 돌고있을 확률
- 어떤 시점에 시스템이 작동 중이고 사용자에게 유용한 서비스를 제공할 확률
- 시스템이 서비스를 요청받았을 때 해당 서비스를 제공할 수 있는 능력

2. 신뢰성 (reliability)

- 특정 시간이 아닌 “기간”
- 주어진 기간 동안 시스템이 사용자가 기대하는 대로 정확하게 서비스를 제공할 확률
- 시스템이 명세화된대로 서비스를 제공할 수 있는 능력
- 정확성(correctness), 정밀성(precision), 적시성(timeliness) 포함 - 295p
- 이 시스템이 잘 죽냐? 안죽냐? 이렇게 생각하면 됨 (고장이 얼마나 자주 나느냐)

  고장이 자주 나면 = 신뢰성이 낮은거 / 고장이 자주 안나면 = 신뢰성이 높은거

3. 안전성 (safety)

- 시스템이 사람 혹은 환경에 손상을 입힐 가능성에 대한 추정
- 시스템이 치명적인 장애 없이 운영될 수 있는 능력

4. 보안성 (security)

- 시스템이 우연적 혹은 의도적 침입을 막아낼 가능성에 대한 추정
- 무결성(integrity), 기밀성(confidentiality) 포함

  무결성은 데이터가 손상되지 않는다 / 기밀성은 데이터가 유출되지 않는다.

5. 복원성 (resilience)

- 시스템이 장비고장, 사이버공격 등 파괴적사건의 존재하에서 중요한서비스를 잘 유지할 수 있는지
- 시스템이 손상 사건에 견디고 복원할 수 있는 능력

예) 1) 가용성이 낮음, 신뢰성이 높음 : 고장이 잘 안나는데 고치는데도 오래걸림

2) 가용성이 높음, 신뢰성이 낮음 : 고장이 자주나는데 고치는데 시간이 짧음

3) 가용성이 높음, 신뢰성도 높음 : 고장은 잘 안나지만 고장이 나도 빨리 고친다.

4) 가용성이 낮음, 신뢰성도 낮음 : 고장은 자주나는데 고치는데 오래걸림

2. 확실성 특성들이 중요한 예시

1. 인슐린 펌프 시스템

   - 중요 : 신뢰성 및 안전성 / 안중요 : 보안성

2. 사람이 없는 황무지에 위차한 기상 시스템

   - 중요 : 가용성 및 신뢰성 (수리 비용이 높기 때문에)

3. Mentcare 환자 정보 시스템

   - 중요 : 보안성 및 복원성 (민감한 개인 데이터를 관리하기에)

3. 확실성에 영향을 미치는 다른 시스템 특성 3가지 Other dependability properties

1. 수리가능성(repairability)

   - 가용성과 관련

   - 시스템이 얼마나 빨리 수리될 수 있는지 => 시스템 장애에 의한 운영 중단을 최소화할 수 있다.

2. 유지보수성(maintainablility)

   - 새로운 요구사항을 수용하기 위한 시스템 변경하여 시스템의 가치를 유지하는 게 중요

3. 오류내성(error tolerance)

   - 사용성과 관련

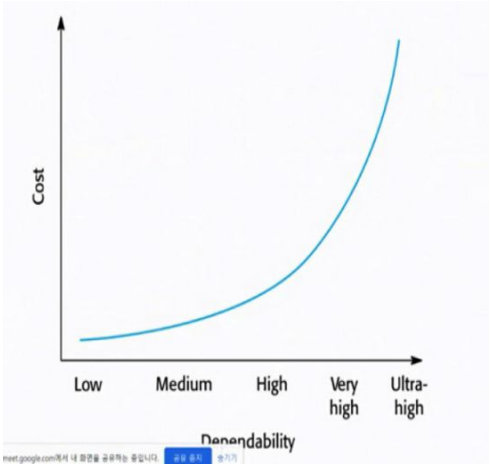
   - 사용자 입력, 조작 오류를 방지하고 감내할 수 있는지의 정도

4. 결함 내성 fault tolerance

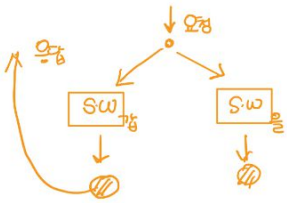
- 결함이 있어도 시스템이 서비스를 계속 제공할 수 있는 능력
- 확실성 잇는 시스템이 자신을 모니터링하고, 잘못된 상태를 감지하고 장애가 발생하기 전에
  결함으로부터 복구하기 위한 코드가 포함하는 것을 의미한다.

5. 확실성과 비용의 관계 곡선 Cost / dependability curve

- 확실성이 있는 시스템을 구축하는 것은 비용이 많이 든다.



10.3 중복성과 다양성 Redundancy and diversity



이렇게 했을때
똑같은 sw니까 하나가 결함이 있으면
다른 하나도 결함이 생길 수 밖에 없어
그래서 하나가 죽으면 다른하나도 같이 죽지
=> 같은 기능을 하되 똑같은 컴포넌트를 쓰면 안
되겠구나 해서
서로 다른 중복 컴포넌트를 사용하자(다형성)

1. 고려사항

- 시스템 컴포넌트의 장애는 피할 수 없음
- 개별 컴포넌트의 장애가 전체 시스템 장애로 이어지지 않도록 시스템을 설계 해야함

2. 중복성과 다양성

- 확실성을 달성하고 향상시키기 위한 전략은 중복성, 다양성 모두에 의존한다.

1. 중복성 redundancy
   - 시스템 컴포넌트의 장애가 일어났을 때 사용할 수 있는 여분의 기능을 시스템에 포함 시키는 것
2. 다양성 diversity
   - 같은방식으로 실패하지않을 확률을 높이기위해 시스템의 중복된컴포넌트들을 서로 다른종류로 사용
3. 확실성을 위해 설계된 시스템은 중복 컴포넌트를 포함한다.
   - 중복 컴포넌트란, 다른 시스템 컴포넌트와 동일한 기능을 제공하는 것
   - 주 컴포넌트(primary,hot)가 실패하면 중복 컴포넌트(backup,standby)로 전환
   - 중복 컴포넌트가 다양하면 중복된 컴포넌트의 공통 결함이 시스템 장애로 이어지지 않는다.

     여럿이서 request를 받고 primary서버가 죽으면 backup하는 컴포넌트가 서비스를 대신 제공

     대신 중복컴포넌트가 다양하지 않고 동일하면 같이 죽을 수 있다 했잖아

     결국 소프트웨어 결함이 같으니까 어떤 부분이 다양성이 있으면 같이 죽지는 않을 것이란 의미
   - 시스템의 상태를 검사하고 복구하는 코드를 포함시킬 수 있다.
   - 이를 통해 장애를 야기하기 전에 문제를 감지할 수 있게 해준다.

3. 중복성과 다양성의 예시

- 가용성이 중요한 요구사항인 시스템에선, 보통 여분의 중복된 서버를 사용한다.
- 이 여분의 서버들은 서로 다른 유형이고 서로 다른 운영체제를 실행하고 있기도 한다.
- 상이한 운영체제를 사용하는 것도 소프트웨어 다양성과 중복성의 예시

  (유사한 기능이 다양한 방식으로 제공)

=> 두 가지 접근법 모두가 안전성이 중요한 상용 시스템에서 이용된다.

4. 문제점

- 중복성과 다양성의 도입에 따라 버그가 생길 수 있음
- 중복성과 다양성은 시스템을 더 복잡하게 만들고 일반적으로 이해하기 어렵게 한다.
- 컴포넌트 장애를 감지하고 다른 컴포넌트로 전환하는 기능이 있어야하므로 시스템의 복잡도 올라감
- 이러한 추가적인 복잡성은 프로그래머가 오류를 만들 가능성을 높이고 시스템을 점검하는 사람이 오류 찾을 가능성을 낮춘다.

=> 복잡도 올라가면 문제생길 확률이 높아져

- 일부 엔지니어들은 중복성과 다양성을 피하는 것이 낫다는 주장도 있음

10.4 확실성 있는 프로세스 Dependable processes

1. 확실성 있는 프로세스의 특징 304-305p

1. 명시적으로 정의된 프로세스 (explicitly definded) <-> 애자일
   - 정의된 프로세스 모델이 있음
   - 개발팀이 프로세스를 준수했다는 데이터가 수집되어야 한다.
2. 반복 가능한 프로세스 (repeatable)
   - 개인의 해석과 판단에 의존하지 않는 프로세스
   - 누가 개발에 참여하는지에 관계 없이 프로세스가 반복될 수 있어야 함

2. 확실성 있는 프로세스의 활동 305-306p

1. 요구사항 검토 2. 요구사항 관리 3. 정형 명세 (수학적)
4. 시스템 모델링 5. 설계 및 프로그램 인스펙션 6. 정적 분석
7. 테스트계획 수립 및 관리

- 시스템 개발 및 테스팅에 초점 맞춘 프로세스 활동뿐만 아니라 잘 정의된 품질 관리 및 변경 관리 프로세스가 있어야 한다.
- 원래 주로 계획 기반 프로세스를 이용하는데 확실성 있는 소프트웨어 개발에 애자일 방법 사용을 고려 -> 교수님 왈 : 아직은 계획 기반으로 하겠지

11장 신뢰성 공학 Reliability Engineering

1. 신뢰성 공학

- 사회의 많은 측면이 소프트웨어 시스템에 의존한다 System reliability
- 중대한 시스템의 경우 고수준의 신뢰성과 가용성이 요구된다.

  > 이 때문에 특수한 신뢰성 공학이 사용

2. 결함-오류-고장 (falut-error-failure) 모델

- Brian Randell이 정의함
- 이 모델은 인간의 오류가 결함을 야기하고, 결함이 오류로 이어지고, 오류가 시스템고장으로 이어지는 개념 기반
- 예시 : 황무지 기상 시스템에는 현재 시간에 1시간을 더하는 방식으로 다음 송신 시간을
  계산한다. 이것이 전송시각 23.00부터 자정 사이에 있을 땐 제대로 작동하지 않는다.
  이유는 24시 시계에서 자정은 00.00이기 때문이다. (사람의 오류)
  전송시간의 변수값이 23.00이상인지 확인하는 검사없이 +1더함 (시스템 결함)
  이때 결함 코드 실행시 전송 시간 변수의 값이 00.xx가 아닌 24.00 (시스템 오류)
  시간이 유효하지 않기 때문에 기상 데이터가 송신이 안됨 (시스템 장애)

1. 사람의 오류 또는 실수 (human error or mistake)

- 시스템 결함이 생기게 하는 사람의 행동

2. 시스템 결함 (system fault) - 버그

- 시스템 오류로 이어질 수 있는 소프트웨어 시스템의 특성

3. 시스템 오류 (system error)

- 사용자가 예기치 못한 시스템 행동으로 이어지는 실행 중의 잘못된 시스템 상태

4. 시스템 장애 (system failure) - 고장,장애

- 사용자가 기대하는 서비스를 시스템이 제공하지 못할때 일어나는 특정 시점의 사건

2. 신뢰성 향상을 위한 세가지 보완적 접근법

- 사람이 실수해서 결함,오류,장애가 생기니까 사람이 실수 안하는 환경을 만들자 (결함회피)

  회피하였지만 그런데도 결함이 일어났을 경우 (결함 감지 및 정정)

  회피하고 찾아 고쳐내도 사실 결함이 존재하니까 (결함 내성)

1. 결함 회피 (fault avoidane)

- SW 설계 및 구현 프로세스는 설계 및 프로그래밍 오류를 피하는 소프트웨어 개발 방식으로 시스템에 도입되는 결함을 최소화해야 한다.
- 강한 자료형 언어 사용, 포인터와 같은 오류가 발생하기 쉬운 요소 사용 최소화

2. 결함 감지 및 정정 (fault detection & correction)

- 검증 및 확인 V&V 프로세스는 프로그램이 운영을 위해 배치되기 전에 결함을 발견하고 제거하도록 설계된다.
- 예 : 체계적인 테스팅, 디버깅, 정적 분석

3. 결함 내성 (fault tolerance)

- 실행 중에 결함이나 시스템의 예기치 못한 행동을 감지하여 시스템 장애가 일어나지 않도록 시스템을 설계
- 결함 내성 시스템 아키텍처

3. 남아있는 결함을 제거하는 비용 (그림11-1)

- 결함을 줄이는게 어렵고 돈이 많이 든다.
- 처음에 결함이 많을때 줄이는건 쉽지만, 결함이 적을때 더 추가로 찾기는 어렵다.
- 소프트웨어의 신뢰성이 더 높아짐에 따라 점점 더 줄어드는 결함을 찾기 위해 더 많은 시간과 노력을 들여야 한다.

11.1 가용성과 신뢰성 Availability and reliability

1. 가용성과 신뢰성 > 계산할 수 있어야됨

1. 가용성 -> 시간

- 서비스가 운영중인지의 여부를 확률로 표현
- 주어진 시점에서 시스템이 운영중이고 요청된 서비스를 제공할 확률
- 시스템 장애 횟수 뿐만 아니라 장애를 야기한 결함을 수리하는데 걸리는 시간에도 의존
- 참고 : 어떤 시점에 시스템이 작동하여 서비스를 제공할 확률
- 예시 : 가용성이 0.999라면 그 시간의 99.9%동안 사용이 가능하다는 것을 의미
- 계산 : 몇번 고장 냈냐 / 전체 시간

2. 신뢰성 -> 기간

- 서비스가 올바른 결과를 제공해주는지의 여부를 확률로 표현
- 주어진 환경에서 특정 목적을 위해 지정된 시간 동안 고장 없이 운영될 확률
- 어떤 특정 시점에 돌고있을 확률 , 고장났을 확률
- 신뢰성은 시스템의 환경에 따라 달라짐 => 사용되는 장소 or 사용하는 방법
- 참고 : 주어진 기간 동안 시스템이 정확하게 서비스를 제공할 확률
- 예시 : 1000번의 입력에 대해 평균 2번의 고장이라면 고장 발생 비율로 표현된 신뢰성은 0.002
- 계산 : 운영된 부분 / 전체 시간

2. 고장 없는 운영 failure-free operation

- 신뢰성에 대한 정의는 고장 없는 운영에 기초한다.
- 고장의 기술적 정의는 시스템 명세를 준수하지 않는 것
- 고장은 객관적 정의가 어려우며 시스템 사용자에 의해 판단됨

  > 이것이 동일한 시스템의 신뢰성에 대해 사용자들이 같은 인상을 가지지 않는 이유

3. 다른 하나보다 더 중요할 때

- 신뢰성과 가용성은 밀접하게 관련되어 있으나, 때로는 어떤 하나가 다른 하나보다 더 중요하다
- 가용성이 더 중요할 때

  : 사용자가 어떤 시스템으로부터 지속적인 서비스를 기대하는 경우 고가용성 요구사항을 가진다. 만약 시스템이 사용자 데이터 손실 없이 장애로부터 빠르게 복구될 수 있다면 이러한 장애는 사용자들에게 큰 영향을 미치지 않을 것이다

=> 전화 교환기

통화 시도할때 통화가 가능하길 기대하기 때문에 가용성 요구사항을 가진다. 연결이 설정되는 동안 결함이 발생해도 신속하게 복구될 수 있고 사용자는 고장이 났는지 모르는 경우가 많다. 만약 통화가 중단이 되어도 심각하게 생각하지 않는다. 왜냐면 중단이 되어도 사용자는 다시 연결을 시도하기 때문이다.

11.2 신뢰성 요구사항 System reliability requirements

1. 신뢰성 요구사항

- 소프트웨어 명세는 불완전하고 부정확할 수 있다.
- 소프트웨어 장애뿐만 아니라 하드웨어 고장과 운영자의 오류도 고려해야 한다.

2. 확실성 요구사항의 2가지 유형 > 대충 넘어가라고 하시고 설명하심

- 시스템 요구사항을 도출할 때 세부적인 것에대한 주의와 시스템 확실성을 보장하기 위한 SW 요구사항 명세가 필요했다.
- 시스템의 전체 신뢰성은 하드웨어 신뢰성, 소프트웨어 신뢰성, 시스템 운영자의 신뢰성에 의해 좌우됨

1. 기능적 요구사항
   - 시스템에 포함되어야 하는 점검 및 복구 기능과 시스템 장애 및 외부 공격에 대한 보호기능 등
2. 비기능적 요구사항
   - 요구되는 시스템의 신뢰성 및 가용성을 정의

   > 신뢰성과 가용성은 시스템의 특성이기에 비기능적 요구사항

11.2.1 신뢰성 척도 Reliabillity metrics

1. 신뢰성, 가용성 3가지 척도

- POFOD와 ROCOF 차이점

: 전자는 잘 일어나지 않고 가끔 일어날때, 요구가 거의 없을 때, 비상시에 사용되는 것

예로 방호 시스템에 대한 요구는 드물지만 0.001이라도 위험한 것으로 여긴다.

: 후자는 가끔일어나는게 아니라 정기적일때 사용

1. 온 디맨드 고장 확률 (POFOD: probability of failure on demand) - 신뢰성 (계산 안냄)

- 요구에 대한 고장 확률
- 요구에 대한 실패가 심각한 시스템 장애로 이어지는 경우
- 요구의 빈도와 관계없이 적용된다.
- 시스템 서비스에 대한 요구가 시스템 장애를 일으킬 확률

2. 고장 발생 비율 (ROCOF: rate of occurrence of failure) - 신뢰성 (계산 낼거임)

- 일반적인 시스템으로 요구의 빈도가 정기적일 때
- 하루에 10번 트랙잭션 실패 or 1000개 트랙잭션 당 10개 실패

1000개의 트랜잭션 중 1개가 실패한다면 고장 확률은 0.001이라 할 수 있다.

- 어떤 시간 간격 (또는 시스템 실행 횟수) 동안 발생할 수 있는 시스템 장애 확률 수
- 고장 간 평균 시간의 역수

2-1. 고장 간의 평균 시간 (MTTF:mean time of failure) - 신뢰성 (계산 낼거임)

- 고장 간의 절대적인 시간이 중요한 경우에 사용한다.
- 긴 트랙잭션을 가진 시스템은 MTTF를 사용
- 평균 트랙잭션 수행 시간보다 길어야 한다 이는, 사용자들이 어떤 세션에도 시스템 장애로 인해

작업한 것을 잃을 가능성이 희박하다는 것을 의미한다.

예시) ROCOF = 12번고장 / 360일 = 0.033.. (360일 동안 12번 고장이 난다)

MTTF = 360일 / 12번 고장 = 30일 (30일마다 고장이 난다)

3. 가용성 척도 (AVAIL:availability) - 가용성 (아마 낼거임)

- 서비스 요구가 있을 때 시스템이 운영 중일 확률 - 가용성 = 운영 시간 / 전체 시간

- 비기능적 신뢰성 요구사항은 신뢰성 척도를 사용 (11.2.2 부분인데 그냥 넘어가심) 324p

11.2.3 기능적 신뢰성 명세 Functional reliability requirements

1. 기능적 신뢰성 요구사항들

- 결함이 발견되고 이러한 결함이 시스템 장애로 이어지지 않도록 보장하기 위해 취해져야할 조치

2. 기능적 신뢰성 요구사항 4가지 유형

1. 검사 요구사항
   - 부정확하거나 범위를 벗어난 입력을 처리되기 전에 감지
2. 복구 요구사항
   - 보통 시스템과 데이터 사본을 관리
   - 장애발생 후 시스템을 복원하는 방법을 명시
3. 중복성 요구사항
   - 단일 컴포넌트 고장이 서비스의 완전한 손실로 이어지지 않는다는 것을 보장하는 시스템의 중복 기능들을 명시
4. 프로세스 요구사항
   - 개발 프로세스에서 좋은 실무관행이 사용되는 것을 보장하는 결함 회피 요구사항

11.3 결함 내성 아키텍처 Fault tolerance

1. 결함 내성

- 결함 : 소프트웨어나 하드웨어에 내재되어있는 특성
- 실행 중에 SW나 하드웨어 고장이 발생하고 시스템 상태가 잘못된 경우에도 시스템이 동작을 계속하기 위한 메커니즘을 포함하는 확실성에 대한 런타임 접근법이다.
- 결함 내성은 결함 오류/ 장애가 시스템 장애로 이어지지 않도록 잘못된 상태를 감지하고 정정함

2. 결함 내성 아키텍처

- 결함 내성을 제공하기 위해 중복되고 다양한 하드웨어 및 소프트웨어를 포함하도록 시스템 아키텍처가 설계되어야 한다.
- 복제 서버를 이용한 아키텍처 328p
  - 두 대 이상의 서버가 동일한 작업을 수행, 서버 관리 컴포넌트가 있음

    확실성이 있는 아키텍처의 간단한 실현은 두 대 이상의 서버가 동일한 작업을 수행하는 복제된 서버들이다. 처리에 대한 요청은 서버 관리 컴포넌트를 통해 특정 서버로 보내진다. 이 컴포넌트는 또한 서버 응답을 추적하고 응답없음으로 감지 가능한 서버 고장이 발생한 경우엔 결함이 있는 서버는 교체된다.
  - 복제 서버는 중복성을 제공하지만 다양성은 제공하지 않음

    복제 서버는 중복성을 제공하지만 일반적으로 다양성을 제공하지 않는다. 서버 하드웨어는 일반적으로 동일하고, 서버는 소프트웨어의 동일한 버전을 실행한다. 따라서 단일 기계에 국한된 하드웨어, 소프트웨어 장애를 대처할 수 있으나, 동시에 소프트웨어의 모든 버전에 장애를 일으키는 소프트웨어 설계 문제에는 대처할 수 없다.

11.4 신뢰성을 위한 프로그래밍

1. 8가지 좋은 실무 지침 > 안나올것 같아서 책 안읽었음 336-342p

1. 정보의 가시성을 제한 - 변수와 데이터 구조에 대한 접근을 통제, 추상 데이터 타입 사용
2. 모든 입력이 유효한지 검사 - 범위 검사, 크기 검사, 표현 검사, 타당성 검사
3. 모든 예외에 대해 처리기를 제공
4. 오류가 발생하기 쉬운 구조의 사용을 최소화 - 부동 소수점 정밀도에 의존, 동적 기억장소 사용
5. 재시작 기능을 제공
6. 자동 배열 경계 검사를 사용
7. 외부 컴포넌트 호출 시 타임아웃 포함
8. 실세계의 값을 나타내는 모든 상수에 이름을 부여

11.5 신뢰성 측정 Reliability measurement

1. 신뢰성 측정에 필요한 데이터는 다음을 포함한다.

1. 서비스 요청 횟수에 대한 시스템 장애 횟수 POFOD
2. 시스템 장애 사이의 시간 간격 또는 트랜잭션 개수 ROCOF, MTTF
   - 시간 단위는 전체 경과 시간 또는 전체 트랜잭션 개수 사용

   => 신뢰성 부분은 결함을 찾기 보다는 신뢰성을 측정하는데 맞춰져 있다.
3. 시스템 장애가 발생한 후 수리 또는 재시작에 걸리는 시간 AVAIL

2. 운영 프로파일 opertaional profile

- 시스템의 입력 유형과 그 입력 유형 발생 확률의 명세로 구성된다.
- 시스템이 실제로 사용되는 방식을 반영
- 새로운 SW 시스템이 기존 시스템을 대체하는 경우에는 사용되는 방식을 이미 알기에 알기 쉬움
- 새롭고 혁신적인 시스템 개발시는 사용되는방식을 예측이 어려워서 정확한 운영파일 만드는게 불가능
- 정확한 운영파일 불가능경우 : 시스템을 사용 방식이 다른 사용자 / 시간이 지나면 사용 방식 변함

> 위와 같은 이유로 신뢰할 수 잇는 운영 프로파일을 개발하는 것은 종종 불가능해, 만약 오래되거나 잘못된 운영 프로파일을 사용한다면 신뢰성 측정의 정확성에 관해 확신할 수 없어.

15장 소프트웨어 재사용 Software reuse

1. 소프트웨어 재사용

- 재사용 기반 소공은 개발 프로세스가 기존 소프트웨어의 재사용을 가능하게 개선된 소공의 전략이다.
- 목적은 높은 소프트웨어 생산성, 저렴한 유지보수 비용, 시스템의 더 빠른 공급, 더 높은 소프트웨어 품질 요구에 대응하기 위해서이다.

2. 재사용 이점

- 신속한 개발, 높은 품질(확실성), 비용 절감, 리스크 감소, 표준 준수 등이 있다.
- 전체 개발 비용이 더 낮아진다는 점이 가장 큰 장점

3. 재사용되는 소프트웨어 단위 4가지

1. 시스템 재사용
    - 여러개의 애플리케이션으로 구성된 전체 시스템 재사용
2. 애플리케이션 재사용
    - 애플리케이션은 변경 없이 다른 시스템에 통합시키거나 재설정하여 재사용
3. 컴포넌트 재사용
    - 단일 객체에서 서브시스템에 이르기까지 다양한 규모의 애플리케이션의 컴포넌트가 재사용
4. 객체와 함수의 재사용
    - 라이브러리 객체 및 함수 재사용

4. 개념 재사용

- 코드를 재사용하는 대신 소프트웨어의 기본적 아이디어, 작업방식, 알고리즘을 재사용
- 예시) 디자인패턴, 아키텍처 패턴 등

5. 재사용 문제점 > 그림 15.2 460p

- 컴포넌트 라이브러리의 생성 및 관리 비용 / 재사용 컴포넌트 검색,이해,변형 / 재사용 컴포넌트의 소스코드가 없으면 유지보수 비용 증가 가능 / 지원 도구 부족 / Not Invented Here 증후군

15.1 재사용 관점

1. 소프트웨어 재사용을 지원하는 접근 방법 4가지 > 462p

1. 애플리케이션 프레임워크
   - 애플리케이션을 생성하기 위한 추상 클래스와 구체 클래스의 집합들이 적용되고 확장된다.
2. 아키텍처 패턴 (서브 시스템 수준)
   - 애플리케이션의 공통적인 유형을 지원하는 표준 소프트웨어 아키텍처
3. 디자인 패턴 (클래스 수준)
   - 추상 클래스와 구체 클래스와 상호작용으로 표현된 일반적인 추상화로 특정 상황에 활용가능
4. 서비스 지향 시스템 -> 18장
   - 외부에서 제공되는 공유 서비스를 연결하여 시스템을 개발

15.2 애플리케이션 프레임워크 Application frameworks

- 배경 : 객체는 다른 시스템에서 재사용될 수 있지만 너무 세분화되는 경우가 많아 특정 애플리케이션에 종속된다는 것을 알게 되었다. 이에 프레임워크가 등장

1. 애플리케이션 프레임워크 > 이것이 뭔지 정도만 알기

- 프레임워크란 큰 크기의 추상개념을 사용하는 객체지향개발 프로세스가 재사용을 가장 잘 지원하더라
- 유사한 형태의 모든 애플리케이션에서 사용될 수 있는 일반화 기능을 제공
- 예) 사용자 인터페이스 프레임워크는 인터페이스 이벤트 처리를 위한 지원을 제공하고, 화면에 표현하기 위해 사용할 수 있는 위젯의 집합을 포함할 것이다.
- 추상 클래스와 구체 클래스의 집합으로 구현되어 있음
  - 따라서, 프레임 워크는 구현 언어에 종속적이다
- 프레임워크 언어는 Java, C#, C++, Python 등 모두 사용 가능하다.
- 애플리케이션을 위한 핵심 아키텍처 제공하여 설계의 재사용을 지원한다.
  - 객체 클래스와 객체 간의 상호작용으로 표현되는 아키텍처, 설계의 재사용

2. 프레임워크의 예

- Graphic User Interface (GUI) 프레임워크

  : 이벤트 처리 기능, 위젯 집합 등 기능 제공

- Web Application Framework (WAF)

  : 보안, 동적 웹페이지, 데이터베이스 통합, 세션관리, 사용자 상호작용 등 기능 제공

3. 프레임워크 확장 Extending frameworks

- 일반적인 프레임워크를 확장하여 구체적인 애플리케이션을 생성
  - 프레임워크가 제공하는 추상 클래스(인터페이스)를 상속받아 구체 클래스를 구현 (추상메소드 구현)
  - 이벤트가 일어나면 호출되는 콜백(callback)메서드 구현

15.3,4 용어만 알기

15-3. 소프트웨어 제품 라인 software product line 467p

- 공통 아키텍처와 일반적인 기능을 제공하는 소프트웨어 애플리케이션 집합
- 일반적인 아키텍처와 공유된 컴포넌트들과 특정 고객의 요구사항을 반영한 특화된 각 애플리케이션을 포함하는 애플리케이션들의 집합이다.
- 새로운 요구사항을 반영하기 위해 몇가지 컴포넌트 설정, 추가적인 컴포넌트의 구현, 몇가지 컴포넌트의 변경들을 포함할 수 있다. 이는 특정 고객의 요구에 맞출 수 있다.
- 물류, 의학 같은 분야에서 도메인 특화된 애플리케이션들을 재사용하기 위해 하드웨어 제어시스템은 이 방법을 자주 사용하여 개발된다.
- 이것은 일반적으로 기존의 애플리케이션들로부터 생겨났다.
  즉, 한 조직이 애플리케이션을 개발하고, 유사한 시스템이 요구될 때, 새로운 애플리케이션에서 개발된 애플리케이션의 코드를 비공식적으로 재사용한다.

15-4. 애플리케이션 시스템 제품

=> 다른 이름 : Commercial Off the Shelf(COTS) 473p

- 판매되는 기성 소프트웨어 시스템
- 일반적인 시장을 위해 시스템 판매자에 의하여 개발된다.
- 애플리케이션 시스템 제품은 시스템의 소스 코드를 변경하지 않고 다른 고객의 필요성에 적응할 수 있는 소프트웨어 시스템이다.
- 애플리케이션 시스템 제품의 다른 이름은 COTS라 하는데 이는 대부분 군용 시스템에서 사용

15-4-2. 설정 가능한 애플리케이션 시스템 Configurable application system

=> 예시 : Enterprise Resource Planning(ERP) 475p

- 특정 비즈니스 유형 또는 비즈니스 전체를 지원하기 위해 설계된 일반적화된 애플리케이션 시스템
- 예 : SAP, Oracle 등에서 제공되는 ERP 시스템
  - 보다 큰 규모에서 보면 대기업에서 전사적 자원 관리인 ERP 시스템은 제조, 주문, 그리고 고객 관계 관리 프로세스를 지원할 수 있다.
  - 이는 주문, 송장작성, 재고관리 그리고 제조 일정관리와 같은 사업 활동을 지원하기 위해 설계된 대규모로 통합된 시스템이다.
  - 이는 회사 도메인에 관계없이 아무데서나 사용 가능해

23장 프로젝트 계획 수립 Project planning

1. 프로젝트 계획 수립

- 소프트웨어 프로젝트 관리자의 가장 중요한 업무 중의 하나

  => WBS : work breakdown structure

- 관리자는 작업을 작은 작업들로 나눔

- 그것들을 프로젝트 팀 멤버들에게 할당

- 발생할 수 있는 문제들을 예측하고 이 문제들에 대한 잠정적인 해결책을 준비해야 함

- 항상 예상치 못한 일들이 일어나기 때문에 요구사항은 변하고 wbs도 달라질테고 일정도 달라지겠지

  -> 그래도 일정을 최대한 지켜달라

2. 프로젝트 생명 주기의 3단계 > 그냥 읽어봐 !

1. 제안 단계
   - 작업을 완수할 수 있는 자원을 가지고 있는지 판단, 제안 가격을 계산
   - 완전한 요구사항이 없으므로 예측에 근거하여 계획 수립
   - 소프트웨어 프로젝트의 제안 가격 측정
     > 인건비, HW와 SW비용, 여비와 교육훈련
     > 추정치를 구한 후 비상 비용을 추가
2. 프로젝트 시작 단계
   - 누가 프로젝트를 작업할지, 프로젝트를 어떻게 작업들로 나눌지, 자원을 어떻게 할당할 지 계획을 수립
   - 요구사항을 더 많이 알고 상세한 계획을 수립해야 함
   - 자원(사람)을 프로젝트에 할당, 새로운 직원의 채용 필요 여부 결정
   - 프로젝트 모니터링 방법을 정의
3. 프로젝트 진행 중
   - 주기적으로 새로운 정보 반영을 위해 계획 수정
   - 요구사항 변경에 따른 작업 분해 구조 변경, 일정 조정
   - 요구사항 변경, 기술 이슈, 개발 문제에 따라 프로젝트 계획은 진화
   - 일정, 비용 추정치, 리스크 등을 갱신

23.1 소프트웨어 가격 책정 Software pricing

1. 가격 책정

- 가격 책정 = 개발 비용
- 프로젝트 비용은 제안서를 근거로 합의한다.
- 상세한 프로젝트 명세서를 확정하기 위해 고객과 공급자 사이에 협상이 진행된다.
- 많은 프로젝트에서 고정된 인자는 프로젝트 요구사항이 아니라 비용이다.

2. Pricing to win

- 뜻 : 합의에 의한 가격 결정이라 불리는 소프트웨어 가격 책정에 대한 접근법
- 고객의 기대 가격을 근거로 계약에 입찰하는 것을 의미

23.2 계획주도 개발 Plan-driven development

1. 계획 주도 기반 개발

- 개발 프로세스가 상세하게 계획된 소프트웨어 공학 접근법이다.
- 문제점 : 소프트웨어를 개발하는 환경 및 사용하는 환경의 변화에 따라 초기의 결정을 수정해야함
- 장점 : 초기 계획 수립으로 직원 가용성, 타 프로젝트 등을 고려할 수 있고 잠재적인 문제와 종속성이 프로젝트 진행 중일 때가 아닌 시작 전에 발견됨.

2. 바람직한 접근법

- 계획 주도와 애자일 개발의 합리적인 혼합
- 프로젝트 유형과 프로젝트 참여자에 따라 달라짐 (대형 - 계획주도 / 소형 - 애자일)

23.2.1 프로젝트 계획

1. 프로젝트 계획

- 프로젝트 계획은 작업을 수행하기 위해 프로젝트에서 이용 가능한 자원, 작업 분할, 일정을 정리한다.
- 계획의 상세한 내용은 프로젝트 조직, 리스크 분석, 작업 분할, 모니터링 등이 포함된다.
- 추가로 형상관리 배치, 유지보수 품질, 검증 계획 등 개발될 수 있다.

23.2.2 계획 수립 프로세스 The planning process

1. 이정표, 산출물

1. 이정표 milestone
   - 진척사항을 측정할 수 있는 일정의 한 지점
   - 간단한 보고서 형태를 가질 수 있음
   - 작업의 진도가 검토될 수 있는 프로젝트 단계의 논리적 끝을 의미
   - 진척 보고를 위한 간단한 보고
   - 테스팅을 위해 시스템을 인도하는 때
2. 산출물 deliverable
   - 고객에게 인도되는 작업 결과물
   - 시스템에 대한 요구사항 문서 또는 시스템의 초기 구현과 같은 좀 더 실질적인 프로젝트 결과물
   - 프로젝트 계획서에 명시됨

2. 프로젝트 계획을 정의

- 낙천적이 아닌 현실적인 가정을 해야한다.
- 프로젝트 진행 중 항상 문제가 발생하고 이것들은 프로젝트 지연이 발생시킨다.
- 초기의 가정과 일정은 비관적이어야하며 예상치 않은 문제들을 고려해야 한다.
- 예상치 못한 문제 발생을 가정하여 인도 일정에 심각하게 영향을 주지 않도록 비상 대책을 세우기
- **리스크 완화 risk mitigation**
  - 상당한 지연에 이르게 할 심각한 문제가 있으면 리스크 완화 행동이 필요하다
  - 프로젝트에 대한 계획 재수립, 고객과의 재협상 등 포함해야 한다.
  - 작업이 완결되어야 할 땐 새로운 일정 또한 수립되어야 하고 고객과 합의를 해야 한다.
  - 만약 리스크 완화가 효과가 없다면, 공식적인 프로젝트 기술 검토를 준비해야 한다.

23.3 프로젝트 일정 관리 Project scheduling

1. 프로젝트 일정 관리

- 프로젝트를 작업으로 구분하여 조직화하고 작업이 실행되는 시점과 작업 실행 방법을 결정하기
- 작업을 실행할 인원과 필요한 SW와 HW 자원들도 추정해야 한다.
- 애자일과 계획 기반 프로세스 모두 초기 프로젝트 일정 필요
  - 〉애자일은 주요 단계에 대한 전반적 일정, 각 단계에서 반복적으로 수립
  - 〉계획 기반은 완전한 일정을 세운 후 프로젝트 진행되면서 수정

2. 계획 주도 프로젝트에서 일정관리

- 작업에 대한 최대 시간은 1~8주 정도로 정하기
  - 〉만약 작업이 이것보다 오래걸리면, 프로젝트 계획 수립과 일정관리를 위해 작은 작업으로 나눠야함
- 작업들 중의 일부는 병행 수행이 가능하게하고 작업 간의 불필요한 종속성을 최소화하게 작업을 편성
- 중대한 작업이 지연되어 전체 프로젝트가 지연되지 않게 하기

3. 프로젝트 일정 관리 문제

- 이전의 프로젝트와 유사한 경우 이전의 추정치를 재사용
- 생산성은 투입된 인력의 수에 비례하지 않음
  - 〉지연된 프로젝트에 인력을 추가로 투입하면 의사소통 오버헤드로 인하여 더 지연 될 수 있다.
- 예상치 못한 일은 항상 일어난다.
- 일이 잘못될 가능성을 반드시 고려해야 한다.
- 새롭거나 기술적으로 진보된 프로젝트는 불확실성이 크다. = 리스크가 크다.

23.3.1 일정 표현 Schedule presentation

〉 시험에 제출 : 그림그리는건 안낼건데 각자 그려서 미룰수 있는날 임계경로 같은거 낼거임

1. 프로젝트 일정 표현 수단

- 액티비티 (= task = 작업) : 프로젝트할때 쪼개는 단위

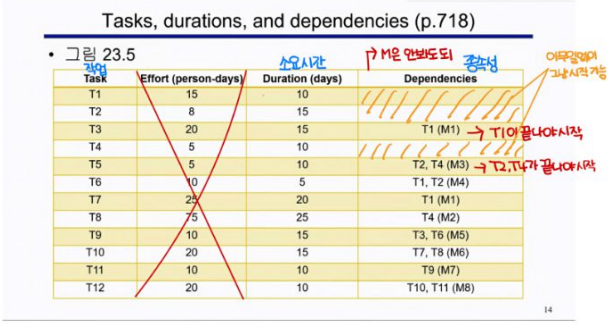
1. 액티비티 네트워크 : 액티비티 간 종속성을 보여줌

2. 바 차트 Gantt chart : 액티비티의 시작/ 종료 시간을 보여줌 , 달력 기반, 간트 차트

2. 액티비티 속성

- 소요 시간 : 보통 1주~8주 사이로 잡아

- 종료점 : 문서, 검토회의, 테스트 완료 등



3. 액티비티 네트워크

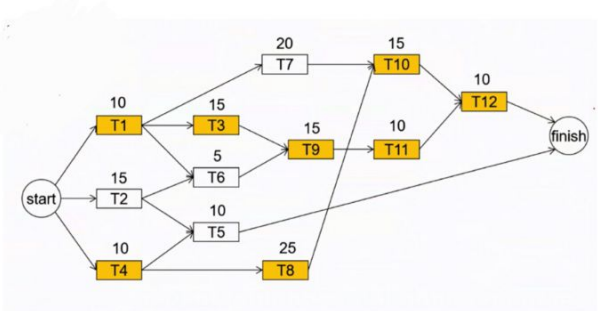
- 노력과 이정표는 고려하지 않기

- 종속성이 없는 작업을 start에 연결

- 종속성은 화살표로 표시하기 / 종속성이 끝나는 부분부터 다음 작업을 시작할 수 있음

- 나가는 화살표 없는 작업을 finish에 연결

- 그림 완성 후 작업 위에 소요시간 적기



4. 액티비티 차트

1. **임계 경로 Critical path**

   - 가장 긴 종속적인 작업들의 순서 => 즉 소요시간이 가장 긴 경로

   - 이 임계경로는 달라지지 않아 (미룰수 없고 항상 같은 위치)

2. 1단계 bar chart (검정 박스)

   - 일에 순서가 있어서 최소 임계시간만큼 걸리는 프로젝트

   - 시간 단축은 어려우나 지연시키지 않게하기 위함

3. 2단계 bar chart (빨간 박스)

   - 임계시간 내에 끝내는 작업 중 최대한 미룰 수 있는 거 찾기

   - finish가 start라 생각하고 화살표가 반대라 생각하고 마지막날이 0이라 생각하고 그리기

4. 3단계 bar chart

   - 지연 가능한 일정을 빗금으로 표시 (분홍 빗금)

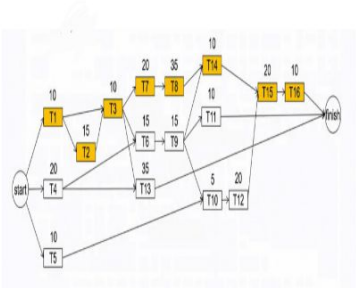
임계 경로 : T1 - T3 - T9 - T11 - T12 (길이 60) or T4 - T8 - T10 - T12 (길이 60)

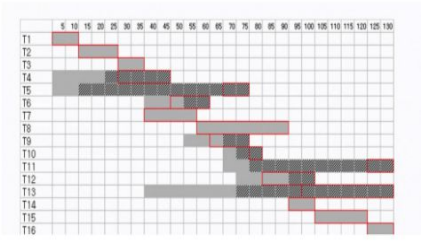
5. 연습문제 23.5

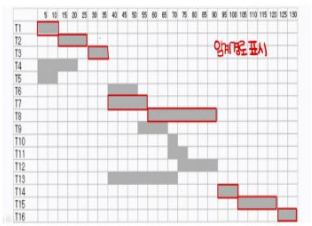
1. 임계 경로를 구하시오 : T1 - T2 - T3 - T7 - T8 - T14 - T15 - T16 (130일)

2. 액티비티 네트워크와 바 차트를 그리시오

3. T4를 가장 일찍 시작할 수 있는 날 : 1일차 / 늦게 시작할 수 있는 날은 26일차





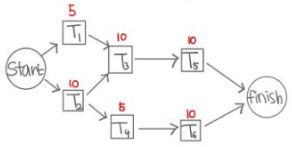


6. 과제 23.5

작업	소요시간(일)	종속성
T1	5	
T2	10	
T3	10	T1, T2
T4	5	T2
T5	10	T3
T6	10	T4

---

(1) 액티비티 네트워크를 그리시오



2017301063 이하연

아이패드 손글씨로 작성

(2) 프로젝트의 최단 수행기간과 임계경로에 속한 작업을 나열하시오

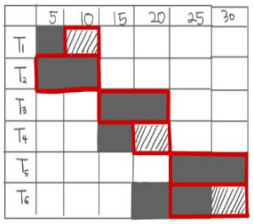
① 최단 수행기간 $T_1 \rightarrow T_3 \rightarrow T_5$ (25일)

$T_2 \rightarrow T_4 \rightarrow T_6$ (25일)

② 임계경로에 속한 작업

$T_2 \rightarrow T_3 \rightarrow T_5$ (30일)

(3) 바 차트 그리기



→ Start부터 화살표를 따라 bar chart 표시

→ finish부터 화살표를 반대로 bar chart 표시

→ 지연가능한 일정을 빗금으로 표시

2017301063 이하연

아이패드 손글씨로 작성

(4) 프로젝트를 지연시키지 않으면서 각 작업을 최대한 미루려고 할 때 각 작업 별로 시작일과 종료일을 나열하시오

	T_1	T_2	T_3	T_4	T_5	T_6
시작일	6일차	1일차	11일차	16일차	21일차	21일차
종료일	10일차	10일차	20일차	20일차	30일차	30일째
	↓	↓	↓	↓	↓	↓
기간	5일	10일	10일	5일	10일	10일

17장 분산 소프트웨어공학 Distributed Software enginerring

1. 분산 소프트웨어 공학

- 최근 대부분의 컴퓨터 기반 시스템이 분산 시스템이다.
- 단일 애플리케이션이 하나의 컴퓨터에서 수행되는 것은 아니고 다수의 컴퓨터들에서 실행되는 것
- 정의 : 사용자에게 하나의 시스템으로 보이는 독립적인 컴퓨터들의 집합

2. 분산 시스템 5가지 장점

1. 자원 공유 (resource sharing)
   - 네트워크로 연결된 컴퓨터들의 하드웨어 및 소프트웨어 자원을 공유
2. 개방성 (openness)
   - 분산 시스템은 일반적으로 오픈 시스템 (표준 인터넷 프로토콜을 중심을 설계된 시스템) 이다.
     > 따라서, 여러 공급업체의 장비와 소프트웨어 사용
3. 동시성 (concurrency)
   - 여러 프로세스들이 네트워크상의 서로 다른 컴퓨터에서 동시에 수행
4. 확장성 (scalability)
   - 시스템의 새로운 수요에 대처하기 위해 새로운 자원을 추가하여 처리 능력을 올림
5. 결함 내성 (fault tolerance)
   - 결함이 발생하여 성능이 저하된 상태에서도 서비스를 제공할 수 있는 능력

3. 분산 시스템의 특징

1. 중앙 집중 시스템보다 더 복잡하다.
   - 이는 설계, 구현, 및 테스트를 더 어렵게 만든다.
   - 시스템 컴포넌트 및 기반 구조 사이의 상호 작용의 복잡성으로 인해 생기는 특성 이해가 어렵다.
     > 예 : 전체 시스템의 성능은 하나의 프로세서의 실행 속도에 의존하기보단,
     네트워크 대역폭, 네트워크 부하, 시스템에 포함된 다른 컴퓨터들의 속도에 더 영향을 받음
   - 분산시스템의 응답시간은 예측하기 어렵다.
     > 응답시간은 전체적인 시스템 부하, 시스템 아키텍처 및 네트워크 부하에 의존적
     > 이러한 모든 요소들이 짧은 시간에 변경가능하므로, 사용자의 서비스 요청마다 응답시간 다름

17.1 분산 시스템

1. 분산 시스템

- 분산 시스템은 그 자체로 복잡한 시스템이고, 이는 네트워크를 사용하는 시스템의 소유자에 의해 제어될 수 없다.

- 따라서 시스템을 설계할 때 분산 시스템에서 본질적으로 예측 불가능한 동작들을 고려해야 한다.

2. 분산 시스템 설계시 고려해야할 6가지 사항

1. 투명성 transparency

   - 사용자에게 어느 수준까지 단일 시스템으로 보여야 하는가

   - 투명성 제공을 위해 시스템이 분산되었다는 것이 사용자에게 감추어져야 함

   - 추상화를 통해 시스템 자원을 숨기면 애플리케이션 변경없이 자원을 이동시키거나 추가할 수 있음

   - 실제 시스템을 모두 투명하게 하는건 불가능하나, 일반적으로 사용자는 분산시스템으로 처리됬음을 알고 있다.

2. 개방성 openness

   - 상호운용성있는 표준 프로토콜을 사용해여 설계되어야 하는가? 특화된 프로토콜을 사용해야 하는가?

   - 표준을 준수하여 컴포넌트를 개발하면 서로 다른 프로그래밍 언어로 구현해도 상호 연동이 가능함

   - CORBA표준, EJB표준 - 웹 서비스 표준, RESTful 프로토콜

3. 확장성 scalability 〉별로 안중요

   - 확장 가능하도록 어떻게 설계할건가?

   - 시스템 확장성은 시스템에 대한 요구 증가에도 고품질의 서비스를 제공할 수 있는 능력을 의미한다.

   - 확장성엔 크기,분산,관리 세가지 중요 요소가 존재 + 수직확장, 수평확장 (별로 안중요)

4. 보안 security 〉별로 안중요

   - 어떻게 개별적으로 관리되는 시스템간의 적용되는 보안 정책을 정의하고 구현할 것인가?

   - 가로채기, 차단, 수정, 위조 공격 유형을 자체적으로 방어해야 함 (별로 안중요)

5. 서비스 품질 quality of service (Qos) 〉별로 안중요

   - 어떻게 시스템 사용자에게 전달되는 서비스 품질의 정의하고 구현할 것인가?

   - 사용자가 납득할 만한 신뢰성 있는 서비스의 전달, 반응 시간 및 시스템의 처리능력을 반영한다.

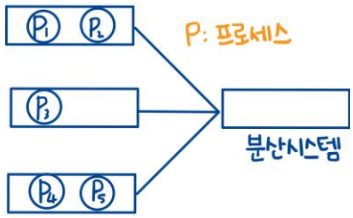
   - 피크 타임 이하의 고품질 서비스를 제공하기 위해 시스템을 설계, 구성하는 것이 비용 대비 효율적

6. 고장 관리 failure management 〉별로 안중요

   - 어떻게 시스템 고장이 감지되고, 컴포넌트 고장 시 비상 운영되고, 수리될 것인가?

   - 분산 시스템에서 장애의 발생은 불가피하기에 장애 발생 시 회복이 될 수 있도록 설계되어야 함

17.1.1 모델의 상호 작용 Models of interaction



1. 분산 시스템의 컴포넌트(컴퓨터) 간 상호 작용

-> 하드웨어가 아닌 프로세스간 상호작용

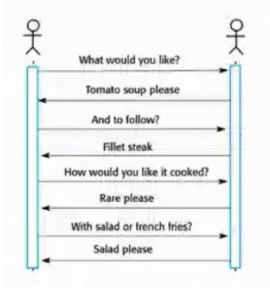
1. 절차적 상호 작용 procedural intercation

- 한 컴퓨터가 다른 컴퓨터에 의해 제공되는 서비스를 호출하고 호출된 서비스가 완료될때까지 기다린다.

2. 메시지 기반 상호 작용 message-based intercation

- 필요한 정보를 메시지 형태로 다른 컴퓨터에 전송하고 응답을 기다리지 않는다.

2. 절차적 통신



* 스텁(stub)은 원격 프로시저의 인터페이스를 정의한다.

* RPC : remote procedure call 원격 프로시저 호출

- 보통 원격 프로시저 호출(RPC)로 구현

- Java RMI : 자바에서는 원격 메서드 호출 RMI들을 원격 프로시저 호출로 볼 수 있다.

- 하나의 컴포넌트는 다른 컴포넌트가 제공하는 서비스를 로컬 프로시저나 메서드처럼 호출할 수 있다

- 시스템 미들웨어는 이러한 호출을 원격 컴포넌트에 전달하고 호출한 컴포넌트에 결과를 반환

  > 1.호출을 원격 컴포넌트에 전달하고 2.필요한 계산을 수행하고 3.호출했던 컴퓨넌트에게 다시 반환

  1,3은 미들웨어가 하는일 / 2는 원격 컴포넌트쪽에서 하는 일

  => 미들웨어는 계산, 서비스제공 하지 않아 단지, 그들 사이의 통신을 도와주는 전달,반환 역할

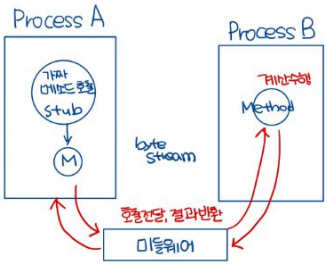
- RPC은 호출을 시작하는 로컬 컴퓨터에서 접근할 수 있도록 호출될 프로세스를 위한 스텁을 요구한다.

- 문제점 : RPC 방식의 상호작용에서는 호출자와 피호출자 모두 통신 시에 참조 가능한 상태여야 하고

  서로를 참조하는 방법도 알아야 한다는 점 (IP주소를 알아야 한다)

- 통신 순서

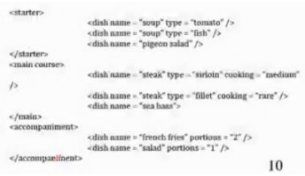
  로컬 컴포넌트가 스텁을 호출하면 매개변수들을 전송 표준 표현으로 변환하여 미들웨어를 통해 원격 프로시저에 요청을 보냄 -> 원격 프로시저는 매개변수들을 필요한 형식으로 변환하여 계산을 수행하고 미들웨어를 통해 결과를 반환 -> 로컬 컴포넌트는 스텁을 통해 반환값을 받는다.



3. 메시지 기반 상호 작용 Message passing

- 일반적으로 다른 컴포넌트로부터 요청된 서비스를 상세하게 정의한 메시지를 생성하는 하나의 컴포넌트를 가진다.
- 절차적 통신의 문제점과 반대로 이 접근법에선 불가용성이 허용될 수 있다.
- 만약 메시지를 처리하는 시스템 컴포넌트를 사용할 수 없는 경우,

  수신자와 온라인 상태가 될 때까지 메시지는 단순히 미들웨어 큐에 남아있게 된다.



- 송신자가 메시지 수신자의 위치나 이름을 알 필요도 없으며 반대의 경우에도 마찬가지이다.
- 송신자와 수신자는 단순히 미들웨어를 통해서 통신한다. => 미들웨어가 메시지의 전달을 보장
- 통신 순서

  서비스에 필요한 상세한 메시지를 생성하여 미들웨어로 전달 -> 미들웨어가 수신 컴포넌트로 메시지를 전송 -> 수신자는 메시지를 해석하여 계산을 수행하고 결과 메시지를 생성 -> 수신 컴포넌트가 결과 메시지를 미들웨어로 전달하면 송신 컴포넌트로 전송된다.

4. 미들웨어 Middleware

- 분산 시스템에서의 미들 웨어

  분산시스템에서의 컴포넌트는 서로 다른 프로그래밍 언어로 구현될 수 있으며, 서로 다른 종류의 프로세서에서 실행 될 수 있다. 이때 데이터 모델, 정보 표현, 통신 규약이 모두 다를 수 있는데 분산 시스템은 이러한 다양한 부분을 관리하고 통신하며 데이터를 교환할 수 있는 소프트웨어가 필요했고 미들웨어가 그 소프트웨어

- 미들웨어는 시스템의 분산 컴포넌트들 사이에 위치 (운영체제와 애플리케이션 사이에서 공통 서비스를 제공)
- 애플리케이션 개발자에의해 특별히 작성된 범용 소프트웨어가 아니며, 일반적으로 구매가능한 기성품 소프트웨어
- 미들웨어 예) 트랜잭션 관리자, 데이터 변환기 등

  => 미들웨어 표준만 지켜주면 C++,자바처럼 프로그래밍언어가 달라도 통신이 가능하다.

4-1. 분산 시스템의 미들웨어는 제공하는 두 가지를 지원

1. 상호 작용 지원
   - 위치 투명성 (location transparency)
   - 다른 종류의 프로그래밍 언어가 사용되었더라도 이벤트 탐지나 통신 등이 있어 파라미터 변환을 지원
2. 공통 서비스의 제공
   - 여러 컴포넌트가 요구하는 재사용 가능한 서비스의 구현을 제공
   - 이러한 컴포넌트들의 기능에 관계없이 다른 컴포넌트에 의해서도 요구될 수 있는 서비스이다.
   - 공통 서비스는 보안, 알림, 네이밍, 트랜잭션 관리 서비스 등을 포함

17.2 클라이언트-서버 컴퓨팅 Client-server computing

1. 클라이언트-서버 컴퓨팅

- 인터넷을 통해 접근하는 분산 시스템은 클라이언트-서버 시스템으로 구성되어있다.
- 클라이언트 : 웹 브라우저, 모바일 앱, 사용자와 상호작용하는 프로그램 (프로세스)
- 서버: 웹 서버 등 서비스를 제공하는 프로그램 (프로세스)

  > 프로세스, 프로세서 구분 : 프로세스는 실행 중인 프로그램 / 프로세서는 컴퓨터

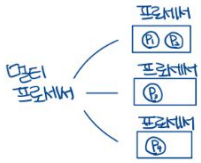
- 클라이언트-서버 시스템에서 사용자들은 그들의 컴퓨터에서 실행되는 웹 브라우저나 모바일 기기에서 실행되는 앱과 상호 작용한다. 이것은 웹 서버와 같은 원격 컴퓨터에서 실행되는 다른 프로그램과 상호 작용한다.
- 클라이언트-서버 아키텍처

  > 애플리케이션은 서버가 제공하는 서비스의 집합으로 모델링된다,

  클라이언트는 이러한 서비스들을 접근하고 최종 사용자에게 결과를 표현할 수 있다.

  클라이언트는 현재 사용 가능한 서버를 알아야 하나 다른 클라이언트에 대해선 몰라도 된다.

- 대부분 서버는 멀티 프로세서 시스템



  - 하나의 서버 프로세스는 여러 프로세서 중 하나에서 실행
  - 서버 프로세스의 인스턴스는 각 CPU에서 수행된다.
  - 부하균형 SW는 클라이언트가 요청한 서비스를 여러서버로 분산하여 각 서버가 같은양의 일을 처리함

2. 분산 클라이언트-서버 시스템의 논리적 4계층

- 계층 간의 분명한 인터페이스를 갖는 4개의 논리적 계층으로 구성되도록 설계해야 한다.
- 이로 인해 각 계층이 다른 컴퓨터로 분산되는 것이 가능해진다.

1. 표현 계층 presentation layer
   - 사용자에게 정보 표현하고 모든 사용자의 상호 작용을 관리한다.
2. 데이터 처리 계층 data-handling layer
   - 클라이언트로부터 전송되는 데이터를 관리 / 웹 페이지 생성 등을 구현
3. 애플리케이션 처리 계층 application processing layer
   - 애플리케이션 논리의 구현과 요구된 기능을 최종 사용자에게 제공한다.
4. 데이터베이스 계층 database layer
   - 데이터를 저장하고 트랜잭션 관리와 질의 서비스(쿼리 서비스)를 제공한다.
   - DBMS

17.3 분산 시스템을 위한 아키텍처 패턴 Architectural patterns

1. 분산 시스템 아키텍처

- 분산 시스템 설계자는 시스템의 성능, 신뢰성, 보안 ,관리용이성 사이의 균형을 찾기 위해 시스템 설계를 구성해
- 분산 애플리케이션을 설계시, 설계자는 시스템에서 요구하는 중요한 비기능적인 요구사항을 지원하는 아키텍처 스타일을 선택해야함

2. 분산 시스템 아키텍처 5가지 스타일

1. 마스터-슬래이브 아키텍처
    - 최소한 응답시간을 보장해야 하는 실시간 시스템에 주로 사용
2. 2단 클라이언트-서버 아키텍처
    - 단순한 클라이언트-서버 시스템을 위해 사용
3. 다단 클라이언트-서버 아키텍처
    - 서버가 대량의 트랜잭션을 처리해야 할 때 사용
4. 분산 컴포넌트 아키텍처
    - 다양한 시스템의 자원과 데이터베이스들이 결합되어야 할 때 사용
5. 피어-투-피어 아키텍처
    - 클라이언트의 로컬 데이터를 공유하고 클라이언트가 서버 역할도 해야 할 때 사용 (클,서 구분이 없음)

17.3.2 2단 클라이언트-서버 아키텍처 Two-tier client server architectures

1. 2단 클라이언트-서버 아키텍처

- 클-서 아키텍처의 가장 단순한 형태로 시스템은 하나의 논리적 서버와 이를 사용하는 다수 클로 구현된다.

2. 2단 클라이언트 2가지 형태 아키텍처 모델

1. 씬 클라이언트 모델 Thin client model
   - 클 : 표현계층만 구현 / 서 : 나머지 계층(데이터제어, 애플리케이션 처리, 데이터베이스) 구현
   - 클라이언트의 표현 소프트웨어는 웹 브라우저나 모바일 기기
2. 팻 클라이언트 모델 fat client model
   - 클 : 표현, 애플리케이션(일부or전체) 계층을 구현 / 서 : 데이터관리와 데이터베이스 기능을 구현
   - 클라이언트 소프트웨어는 서버 애플리케이션과 밀접한 관련이 있는 특별히 작성된 프로그램이 될수있다.

3. 씬 클라이언트

1. 장점은 클라이언트 관리가 간단하다. -> 다수의 클라이언트가 존재할때 효과적
2. 단점은 서버와 네트워크 모두에 큰 부하가 발생한다.

4. 팻 클라이언트

- 클라이언트 소프트웨어를 실행하는 컴퓨터에서 사용 가능한 처리 능력을 활용
- 클라이언트 컴퓨터에 소프트웨어를 배포하고 유지하기 위한 시스템 관리가 필요하다.

=> 모바일 기기 사용의 증가로 네트워크 트래픽을 최소화하는 것이 중요해졌는데

결과적으로 씬,팻 클라이언트 아키텍처 사이의 구분은 무의미해지고 있다.

17.3.3 다단 클라이언트-서버 아키텍처 Multi-tier client-server architectures

1. 2단 클라이언트-서버 아키텍처의 문제

- thin 클라이언트는 확장성과 성능의 문제가 발생할 수 있고
- fat 클라이언트는 시스템 관리의 문제를 일으킬 수 있다.

=> 이를 회피하기위해 다단 클라이언트-서버 아키텍처가 등장

2. 다단 클라이언트-서버 아키텍처

- 시스템의 다양한 계층들(표현, 어플리케이션, 데이터관리, 데이터베이스)이 다른 프로세서에서 실행될 수 있는 별도의 프로세스들이다.
- 3단 클라이언트-서버
  - 서버를 추가한 형태
  - 데이터 관리를 위해 웹 서버를 사용할 수 있게하고, 애플리케이션 처리와 데이터베이스 서비스 서버의 분리가 가능하게 한다.
- 다단 클라이언트-서버 (이걸 많이 사용)
  - 여러 대의 서버를 통해 애플리케이션 처리를 분산하므로 2단보단 확장성이 더욱 뛰어난다.
  - 수 많은 클라이언트가 있는 애플리케이션에 적합하다.

17.3.4 분산 컴포넌트 아키텍처 Distributed component architectures

1. 분산 컴포넌트 아키텍처

- 등장 배경? 애플리케이션의 다양한 유형에 잘 동작하나 각 서비스가 포함되어야 할 계층을 선택함에 있어서는 어느정도 시스템의 유연성을 제한하고 있어서 설계자들은 확장에 대한 대비를 하며 설계해야하고 추가되는 클라이언트를 언제나 수용할 수 있도록 서버 증설을 위한 방안도 수립해야 한다.
- 일반적인 접근 방법은 시스템 계층들에 이러한 서비스를 할당하지 않고 시스템을 몇개의 서비스들로 설계하기
- 상호작용(서비스 제공/사용)하는 컴포넌트들의 집합으로 구성
- 각 컴포넌트는 그들이 제공하는 서비스에 대한 인터페이스를 제공
- 다른 컴포넌트들은 RPC나 메서드 호출을 사용해 미들웨어를 통해 서비스를 호출

2. 분산 컴포넌트 미들웨어

* CORBA 표준은 분산 컴포넌트 시스템에 대한 미들웨어를 정의 but CORBA 구현은 실제로 많이 못 사용

- 분산 컴포넌트 시스템들은 미들웨어에 의존적이다.

- 컴포넌트 간 상호작용을 관리하고 전달된 파라미터의 유형들의 차이를 조정하고 공통 서비스의 집합을 제공한다.

- CORBA(Common Object Request Broker Architecture)

3. 분산 컴포넌트 모델의 장단점

* SOA : 서비스 지향 아키텍처(Service-Oriented Architecture:SOA)

1. 장점

  - 서비스가 어디서, 어떻게 제공되어야 하는가에 대한 결정을 연기 가능

  - 새로운 자원을 추가하는 것을 허용하는 매우 개방적인 시스템 아키텍처이다.

    즉 신규 시스템 서비스들은 기존 시스템에 큰 혼란 없이 쉽게 추가 가능하다.

  - 유연하고 확장성이 높다. 시스템의 다른 부분에 영향을 주지 않고 시스템 부하가 증가함에 따라

    새로운 객체 또는 복제된 객체들이 추가될 수 있다.

  - 필요에 따라 네트워크를 통해 컴포넌트를 이동시켜 시스템을 동적으로 재구성하는 것이 가능하다.

2. 단점

  - 클라이언트-서버 시스템에 비해 더 복잡하다.

  - 분산 컴포넌트 모델이나 미들웨어에 대한 보편적인 기준이 없다.

  => 이 문제점의 결과로 이 모델은 **SOA(서비스 지향 시스템)**으로 대체되고 있다.

    하지만 분산 컴포넌트는 SOA에 비해 성능상의 장점이 있다 (RPC통신 빠르다)

    그래서 아직 많은 수의 트랜잭션들을 신속하게 처리해야하고 높은 처리량이 요구되는 시스템에선 사용해

17.3.5 피어-투-피어 아키텍처 Peer-to-oeer architectures

1. 피어투피어 아키텍처

- 클-서모델은 서버와 클라이언트가 명확하게 구분되나 서버가 더 많은 작업을 하게되어 부하가 편중되는 문제발생
- 피투피 모델은 네트워크의 어떠한 노드도 주어진 연산을 수행할 수 있는 비중앙집중적인 시스템이다.

  즉, 중앙집중이 아니고 모두 똑같다
- 서버와 클라이언트의 구분이 없다
- 네트워크 상에 존재하는 수많은 컴퓨터의 자원(저장공간, 연산능력)의 장점을 활용하기 위해 설계되었다.
- 대부분 비즈니스 시스템보다 개인적 목적으로 사용되어 왔다.

  〉 중앙서버가 x : 피투피 시스템을 모니터하고 관리하기 어렵다는 것을 뜻하고 더 높은 수준의 개인 통신이 가능
- 저장공간 예시 예시 : Bitcoin, BitTorrent / SETI@HOME은 grid computing의 일종

2. 피어투피어 아키텍처의 종류

1. 완전 비중앙집중 p2p 아키텍처
   - 원칙적으로 피투피 네트워크에 존재하는 모든 노드들은 모든 다른 노드를 알 수 있어야 한다.
   - 각 노드들은 다른 어떤 노드와도 연결될 수 있고 직접적으로 데이터를 교환할 수도 있다.
   - 고도로 중복된 장점을 가지고 있어, 네트워크 연결 손실에 대한 내성과 결함 내성을 갖는다.
2. 반-집중 p2p 아키텍처 (슈퍼 피어 super peer) semicentralized
   - 네트워크에 존재하는 하나 이상의 노드들이 노드 간의 통신을 담당하는 서버 역할을 한다.
   - 이 아키텍처는 서버의 역할은 피어 간의 연결 형성을 돕거나 연산의 결과를 조정하는 것
   - 슈퍼 피어 역할 : 작업을 다른 노드에게 분산시키고 연산의 결과를 수집하고 점검하는 것

17.4 서비스로서의 소프트웨어 Software as a service

1. SaaS의 주요 요소

1. 소프트웨어는 서버(거의 대부분 클라우드)에서 실행되고 웹 브라우저를 통해 접근할 수 있다.

   > 이것은 로컬 PC에 설치되지 않는다.

2, 소프트웨어는 소프트웨어 제공자가 소유하고 관리한다.

3. 사용자는 실제 사용량이나 월별 및 연도 별로 서비스를 신청(구독)하여 소프트웨어 비용을 지불할 수 있다.

4. 예시) Google Docs, Office365 등

5. SaaS 발전은 클라우드 컴퓨팅의 보급으로 인해 급격하게 증가하고 있다.

2. SaaS와 SOA 차이점

1. SaaS 서비스서의 소프트웨어

- 웹 브라우저를 통해 접근하는 클라이언트에게 원격 서버가 기능을 제공하는 방법 중 하나
- 서버는 상호 작용을 위한 세션 동안, 사용자의 데이터와 상태를 유지한다.
- 예) 문서 편집과 같은 긴 트랜잭션 작업
- 애플리케이션 기능을 사용자에게 제공

2.SOA 서비스 지향 아키텍처

- 소프트웨어 시스템을 별도의 상태가 유지되지 않는 서비스의 집합을 구조화하는 방법
- 서비스는 다중 제공자에 의해 제공되며 분산될 수 있다.
- 트랙잭션은 짧으며 서비스가 호출되고 작업을 수행하고 그 결과를 반환하는 형태이다.

  (서비스 호출-작업수행-결과 반환하는 짧은 트랜잭션)
- 애플리케이션 시스템을 구현하는 기술

18장 서비스지향 소프트웨어공학 Service-oriented Software Engineering

1. 웹 서비스 Web services

- 정의 : 다른 프로그램이 사용 가능하게 한 계산 및 정보 자원을 위한 표준 표현이다.
- 다른 정의 :

  이산적인 기능을 캡슐화하여 느슨하게 결합된, 재사용가능한 소프트웨어 컴포넌트로, 분산되어 있고 프로그램으로 접근할 수도 있다. 웹 서비스는 표준 인터넷과 XML기반 프로토콜을 이용하여 접근하는 서비스이다.

- 서비스는 서비스를 사용하는 애플리케이션 독립적

  => 만약 "세계지도"라는 서비스가 있다면 "세계지도"를 지원해주는 어플리케이션 A,B,C와 다 독립적이고 A에 세계지도 서비스 부분이 망가져도 다른 세계지도를 쓰는 앱과는 상관없어.

- 서비스 제공자는 조직 외부의 사용자에게 서비스를 제공
- 웹 서비스는 다양하고 일반적인 서비스의 한 사례이다.
- 컴포넌트 기반 소공에서의 서비스와 소프트웨어 컴포넌트의 중요한 차이는 서비스가 독립적이어야 하고 느슨하게 연결되어야 하고 외부 컴포넌트에 의존하지 않아야 한다.
- 웹 서비스 인터페이스는 단순히 서비스 기능과 매개변수를 정의하는 서비스 제공 인터페이스이다.

2. 서비스 지향 시스템

- 독릭적 서비스인 분산 시스템을 개발하는 방식이다.
- 서비스들은 플랫폼 및 구현 언어에 독립적이다.
- 재사용 가능한 서비스 컴포넌트를 이용하여 구현되고, 사용자가 직접 접근이 아니라 다른 프로그램이 접근한다.
- !!구분) 서비스로서의 소프트웨어 :

  소프트웨어 기능을 웹을 통해 원격으로 제공한다는 의미

2. 서비스 지향 접근법의 장점

- 서비스는 조직 내부나 외부의 서비스 제공자로부터 제공될 수 있다.
- 서비스에 대한 정보가 공개되어 있으며 권한이 있으면 누구나 사용 가능하다.
- 애플리케이션은 배치 혹은 실행 시까지 서비스 바인딩을 지연할 수 있다.
- 새로운 서비스의 편리한 구축이 가능함

  서비스 제공자는 혁신적인 방식으로 기존의 서비스들을 연결하고 만들 수 있는 새로운 서비스들을 인식할 수있다.

- 서비스 사용자는 (서비스 제공이 아닌) 사용에 따른 비용을 지불 가능하다.
- 많은 계산이나 예외 처리는 외부 서비스에 떠넘겨 애플리케이션이 더 소형화 됨 (경량화 됨)

- 들어가기 전 배경

- 서비스 제공 및 구현의 초기 작업은 소프트웨어 산업의 컴포넌트 표준에 대한 합의 실패에 큰 영향을 받음
- 따라서 표준이 중요해졌고, 그로 인해 서비스 지향 아키텍처의 모든 표준 및 개념으로 귀결됐어.
- 아키텍처가 서비스 기반 시스템으로 제안되었으며, 모든 서비스 통신 또한 표준 지향적으로 제안되었어.
- 그러나 제안된 표준(XML web service)은 복잡했고 상당한 실행 오버헤드를 가지게 되었는데
- 이 문제 때문에 RESTful 서비스에 대한 대체 가능한 아키텍처 접근방식을 적용하게 만든거야
- 이 접근 방식은 서비스 지향 아키텍처보단 간단하나, 복잡한 기능을 제공하는 서비스에는 부적절해

18.1 서비스 지향 아키텍처 Service-oriented architectures > 가장 중요

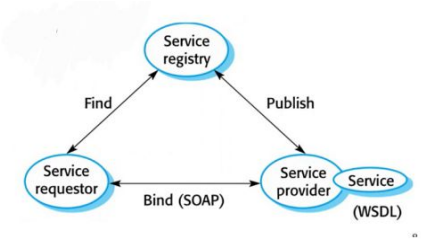
1. 서비스 지향 시스템

- 느슨하게 연결된 아키텍처를 가지고 있는데, 이 아키텍처에서는 시스템 수행 중에 서비스 바인딩이 변경되기도해

  > A에서 서비스를 받다가 서버나 네트워크가 문제가 생기게 되서 못사용하면 같은 종류의 서비스를 제공하는 곳에 가서 서비스를 이용하면 된다.

2. 서비스 지향 아키텍처 SOA

- 실행 가능한 서비스를 애플리케이션에 두는 것을 기반으로한 아키텍처 스타일이다.
- 서비스는 잘 정의되고 공개된 인터페이스를 가지고, 애플리케이션은 이들을 적절성 여부에 따라 선택할 수 있어.
- 다른 공급자들도 같은 서비스를 제공하고, 애플리케이션이 서비스 제공자에 관한 실행 시간에 따른 결정 내림
- XML로 표현되는 메시지 교환에 의해 통신하고 이 메시지들은 HTTP,TCP/IP 같은 인터넷 프로토콜을 이용해 분산됨
- 서비스 지향 아키텍처 구조



  - 서비스 제공자 : 서비스를 설계 및 구현하고 이 서비스들에 관한 인터페이스를 명세한다.
    또한 접근 가능한 레지스트리에 이러한 서비스에 관한 정보를 공개해놓는다.
  - 서비스 요청자 : 서비스 명세를 검색하여 서비스 제공자의 위치를 찾고
    표준 서비스 프로토콜을 사용하여
    특정 서비스를 애플리케이션에 연결(bind)하여 통신할 수 있다.
  - 서비스 레지스트리

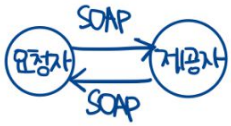
3. SOA 장점

- 외부 제공자에게 서비스를 아웃소싱 가능

- 서비스는 언어 독립적

- 단순화된 정보 교환을 통해 조직 간의 컴퓨팅이 가능

3. SOA의 핵심 표준들 key standards



1. SOAP (Simple Object Access Protocol) - 중요

   - 서비스 메시지(정보) 교환을 위한 표준

   - 서비스 사이의 통신을 지원하는 메시지 교환 표준

   - 서비스 간에 전달되는 메시지의 필수적인 컴포넌트와 선택적인 컴포넌트를 정의

   - 서비스 지향 아키텍처에서의 서비스는 종종 SOAP 기반 서비스라고 불린다.

2. WSDL (Web Service Description Language) - 중요

   - 웹 서비스 정의 언어는 서비스 인터페이스 정의를 위한 표준

   - 서비스 오퍼레이션 (오퍼레이션 이름, 매개변수, 타입)과 서비스 바인딩이 정의되는 방식을 결정

   - 웹 서비스를 이용하고자 한다면 서비스 인터페이스의 세부사항들을 알 필요가 있다

   세부사항들은 WSDL로 불리는 XML 기반 언어로 서비스 설명에 나와있다.

   => 어떤 서비스를 하고, 서비스가 어떻게 통신하는지, 그리고 서비스를 어디에서 찾는지 정의

   (WSDL안에 이와 같은게 있다는 정도만 알기)

   1) 인터페이스 (what) : 서비스가 지원하는 오퍼레이션을 명세하고 서비스가 송수신하는 메시지 형식을 정의

   2) 바인딩 (how) : 추상적 인터페이스를 구체적인 프로토콜 집합으로 변환한다.

   3) 구현 위치 (where) : 특정 웹 서비스 구현 위치(종료점 포함)를 기술한다.

3. WS-BPE - 하지마

4. UDDI - 하지마

18.2 RESTful 서비스 RESTful services

> 이런게 있구나 정도, 시험 낼지 안낼지 모르겠음, 개념정도만! 깊게 들어가지 말기

1.배경

웹서비스와 서비스 지향 소공의 초기 개발은 XML 기반 서비스 간 메시지 교환에 의한 표준 기반이었는데 문제는 이 웹 서비스 표준이라는 것이 아주 두껍고(무겁고) 종종 지나치게 일반적이거나 비효율적이야. XML 메시지와 관련해서 상당한 양의 생성, 전송, 해석들이 이를 실행하기 위해 필요했어, 이는 서비스 간의 통신을 느리게 했고, 대용량 시스템에선 추가적인 요구 서비스 품질을 맞추기 위해 추가적인 하드웨어가 요구되었어. 이와 같은 상황을 대처하기 위해서 대안으로 "가벼운" 방식의 웹 서비스 아키텍처가 개발되었어. 그게 RESTful 이다.

2.RESTful 서비스

- 서버에서 클라이언트까지의 자원의 전송표현을 기반으로 하는 아키텍처 스타일이다.

- 이는 웹 서비스 인터페이스를 구현하기 위한 방법으로 SOAP/WSDL보다 더욱 간단한 방법으로 사용되고 있고, 웹 전체의 기저를 이룬다.

- RESTful는 오버헤드가 적은 서비스

- RESTful의 기본 요소는 "자원"이고 RESTful 아키텍처에선 모든 것이 자원으로 표현될 수 있다.

- JSON의 사용(간단) -> 웹 서비스는 XML을 사용하여 오버헤드가 크다(길다)

- 자원

  > 문서에 해당하는 데이터 요소 > URL처럼 각기 다른 식별자가 있다.

  > 웹은 이 아티켁처를 가진 시스템의 한 예인데, 웹 페이지가 자원이고 웹페이지의 유일한 식별자는 URL이다.

3.자원 오퍼레이션

- Create : 자원을 내부로 가져온다 (생성) => POST 자원 생성

- Read : 자원의 표현을 반환한다. (읽기) => GET 자원값 읽기

- Update : 자원값을 바꾼다. (변경) => PUT 자원값 갱신

- Delete : 정보를 접근불가능하게 만든다. (삭제) => DELETE 자원삭제

2.RESTful 문제점

- 서비스가 복잡한 인터페이스를 가지고 있고 자원이 단순하지 않을 경우 인터페이스에 RESTful 서비스를 이용하여 설계하는 것이 어려움

- 공식적인 RESTful 서비스 표준이 없기에 비공식적 문서에 의존해야 함

- 서비스 품질이나 신뢰성을 위한 구조를 직접 구현해야 한다.

=> 요즘엔 Microsoft, Google, Amazon과 같은 클라우드 서비스 제공자들은 얘와 SOAP API을 같이 사용

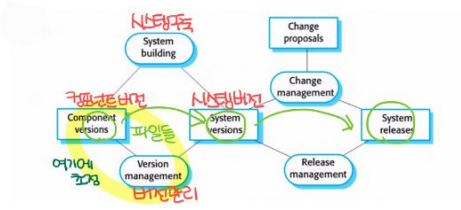
25장 형상 관리 Configuration management

1. 형상 관리 CM

- 소프트웨어 시스템은 개발과 사용 중에 끊임없이 변경된다.
    - 그러면 오류들이 발견되고 수정되어야 한다.
    - 시스템 요구사항이 변경되고 시스템의 새로운 버전에서 이런 변경이 구현되어야 한다.
    - 만약 플랫폼의 새로운 버전이 나오면 그에 맞춰 소프트웨어 시스템을 변경해야 한다.
    - 새로운 기능이 도입하면 그것에 대해 대응해야 한다
    - 그 변경이 소프트웨어에 반영되면, 시스템의 새로운 버전이 생성된다.

    => 따라서 시스템은 버전들의 집합으로 생각할 수 있고, 각각의 버전들은 유지되고 관리되어야 한다.
- 형상 관리는 변화하는 소프트웨어 시스템을 관리하기 위한 정책, 프로세스, 그리고 도구들과 관련
    - 각각의 시스템 버전에 어떤 변경들 or 컴포넌트 버전들이 포함되었는지를 놓치기 쉽기에 관리가 필요
    - 개발 팀에 개발중인 시스템에 대한 접근을 제공하고 코드에 수행한 변경들을 관리한다.
    - 여러 개발자들이 동시에 작업하는 팀 프로젝트에서는 필수적이다.
- 한 사람이 처리해야 하는 변경을 잊기 쉬우므로 개별 프로젝트들에 유용하다.

2. 형상 관리 4가지 활동



1. 버전 관리 > 중요
    - 컴포넌트(파일 소스코드)의 버전을 관리
    - 컴포넌트들의 여러버전들을 추적 관리하고 여러개발자들에 의한 컴포넌트 변경이 서로 방해받지 않도록 보장
2. 시스템 구축 > 안중요
    - 컴포넌트와 라이브러리 등으로 프로그램 생성
    - 프로그램 컴포넌트, 데이터, 라이브러리들을 조립하고 실행가능한 프로그램을 생성하기 위해 컴파일,링킹함
3. 변경 관리 > 안중요
    - 인도된 소프트웨어에 대한 고객과 개발자의 변경 요청을 추적 관리
    - 이런 변경을 구현할 것인지, 한다면 언제할것인지를 판단하는 것과 관련
4. 릴리스 관리 > 안중요
    - 외부 릴리스를 위해 소프트웨어를 준비하고 고객 사용을 위해 릴리스되는 시스템 버전들을 추적 관리

3. 형상 관리 도구

1. 배경 : 형상 관리에서는 관리되어야 하는 정보와 형상 항목들 사이의 관계가 많다

   => 도구없이 할 수 없다. (도구 지원이 필수적)

2. 형상 관리 도구 기능

   - 시스템 컴포넌트들의 버전들을 저장하고

   - 이 컴포넌트들로부터 시스템을 구축하고,

   - 고객에 대한 시스템 버전의 릴리스를 추적하고,

   - 변경 제안들을 추적하기 위해 사용된다.

3. 형상 관리 도구 사용 방식

   - 컴포넌트 확정적인 버전은 공유 저장소에 보관됨

   - 개발자들은 공유 저장소에서 개인 작업 공간으로 복사하여 수정

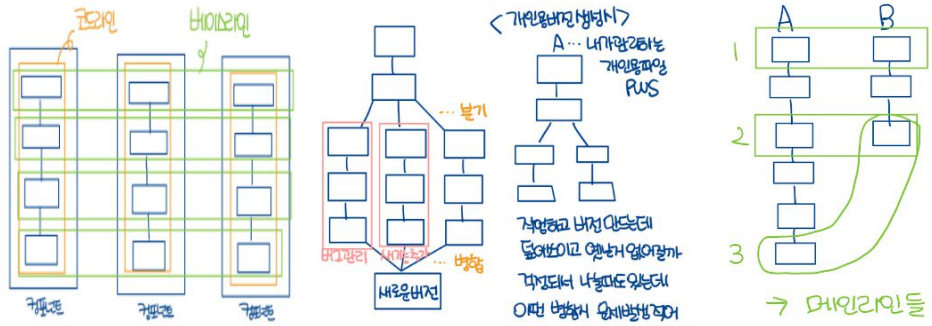
   - 변경 작업이 만족스러우면(변경됨), 수정된 컴포넌트들을 프로젝트 저장소(공유저장소)에 반환

   - 변경된 컴포넌트를 저장소에 기록하면 다른 개발자가 이용할 수 있음

4. CM도구들을 사용하지 않으면 컴포넌트와 시스템이 하루에 여러 번씩 변경되는 애자일 개발은 불가능하다.

4. 형상 관리 용어

용어	설명
베이스라인 baseline	시스템을 구성하는 컴포넌트 버전들의 모음. 베이스라인들은 통제되는데, 이것은 베이스라인에서 사용되는 컴포넌트 버전들은 변경될 수 없다는 것을 의미. 베이스라인은 그것의 구성 컴포넌트들로부터 항상 다시 생성될 수 있다.
분기 branching	기존의 코드라인의 한 버전에서 새로운 코드라인의 생성. 새로운 코드라인과 기존 코드라인은 독립적으로 개발 될 수 있다.
코드라인 codeline	소프트웨어 컴포넌트 그리고 컴포넌트가 의존하는 다른 형상 항목들의 버전들의 집합 (한 소스코드 파일의 버전들)
형상(버전) 관리 configuration/ version control	시스템의 수명 동안 변경들은 관리되고 컴포넌트들의 모든 버전들은 식별되고 저장되도록 시스템과 컴포넌트들의 버전들이 보관되고 유지되는 것을 보장하는 프로세스
형상 항목 또는 SW 형상 항목 software configuration item	형상 제어를 받는 소프트웨어 프로젝트와 관련된 모든 것(설계, 코드, 테스트 데이터, 문서 등), 형상 항목은 유일한 식별자를 가짐
메인라인 mainline	시스템의 여러 버전들을 나타내는 일련의 베이스 라인들
합병 merging	서로 다른 코드라인의 분리된 버전들을 병합하여 새로운 버전을 생성, 이 코드 라인들은 일반적으로 분기에 의해 생성된 것임
릴리스 release	사용을 목적으로 고객에게 인도하는 시스템의 버전
저장소 repository	컴포넌트 버전들이 여기에 쭉 있다는 정도만 알기
시스템구축 system building	적절한 컴포넌트 버전과 라이브러리를 컴파일하고 링크하여 실행 가능한 시스템 버전을 생성
버전 version	형상 항목의 다른 인스턴스(instance)와 구분되는 인스턴스, 버전은 유일한 식별자를 가짐
작업공간 workspace	소프트웨어를 사용하거나 수정하고 있는 다른 개발자에게 영향을 미치지 않고 소프트웨어를 수정할 수 있는 개인 작업 공간



- 베이스라인과 코드 라인을 구분해야되

- 분기는 여러 목적들이 있는데 보통 독립적으로 관리하거나 개인용으로 가져가거나 버그 수정을 위함

- 병합시, 충돌발생 가능성이 있는데 우리는 이를 해결해야 되 (개인용은 병합시 문제발생 적어)

25.1 버전 관리 Version management

1.버전 관리

- 컴포넌트들 그리고 이 컴포넌트들이 사용되는 시스템의 여러 버전들을 추적 관리하는 프로세스이다.

- 버전 관리는 코드라인과 베이스라인을 관리하는 프로세스이다. (메인라인- 시스템의버전)

2.코드라인, 베이스라인 차이

1. 코드라인

    - 소스코드의 일련의 버전들

    - 나중 버전들은 이전의 버전들로부터 유도된다.

    - 시스템의 컴포넌트들에게 적용된다.

2. 베이스라인

    - 특정한 시스템에 대한 정의이다.

    - 시스템에 포함되는 컴포넌트 버전들을 명시하고 사용된라이브러리들, 설정 파일들, 다른 시스템 정보를 식별

    - 즉, 각 개발단계에 있어서 한 개발단계의 승인된 산출물

3. 버전 관리 시스템의 두가지 유형 Version control systems

- 버전 제어 시스템은 컴포넌트의 버전을 저장하고 식별하고 접근을 제어

1. 중앙 집중 시스템

    - 하나의 마스터 저장소에 컴포넌트의 모든 버전을 유지

    - CVS , Subversion

2. 분산 시스템

    - 컴포넌트 저장소의 여러 버전이 동시에 존재

    - Git

4. 버전 관리 시스템의 주요 기능

1. 버전과 릴리스 식별

   - 컴포넌트의 관리되는 버전들은 시스템에 제출될 때 유일한 식별자를 할당 받는다.

2. 변경 이력 기록

   - 컴포넌트들에 변경 내용을 나타내는 태그를 붙이는 것을 포함한다.

   - 변경에 대한 기록을 유지

3. 독립적 개발 지원

   - 다른 개발자들이 동시에 같은 컴포넌트에 대해 작업할 수 있다.

4. 프로젝트 지원

   - 버전 관리 시스템은 컴포넌트들을 공유하는 여러 개의 프로젝트 개발을 지원할 수 있다.

   - 모든 파일들을 체크인, 체크아웃하는 것이 대체로 가능하다.

5. 저장공간 관리

   - Delta를 이용한 컴포넌트 버전의 효율적 저장

   - 동일한 파일들에 대한 중복된 복사본을 유지하는 효율적인 기법을 사용할 수 있다.

5. 모든 버전 관리 시스템

> 체크인,체크아웃 시험에 출제

- 방해가 없는 독립적인 개발을 지원하기 위해,

  모든 버전 관리 시스템은 프로젝트 저장소와 개인 작업공간이라는 개념을 사용한다.

- 프로젝트 저장소는 모든 컴포넌트들의 마스터 버전을 유지한다.

- 컴포넌트를 수정할땐, 개발자는 이것을 저장소에서 자신의 개인 작업공간으로 복사하고,

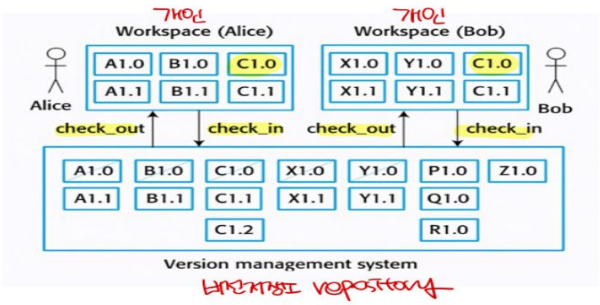
  이 복사본에 대한 작업을 수행 => Check out

- 개발자가 변경 작업을 완료하면, 변경된 컴포넌트들을 저장소로 반납한다. => Check in

- 그러나 중앙집중 시스템과 분산 시스템은 공유된 컴포넌트의 독립적인 개발을 다른방식으로 지원한다.

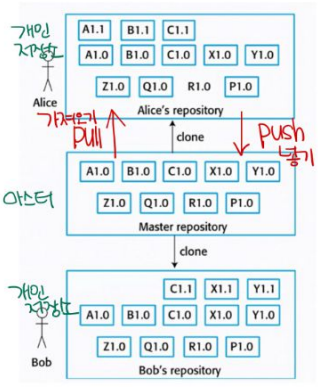
5. 중앙 집중 버전 제어 Centralized version control

- 체크아웃 -> 작업 -> 체크인(작업한 내용을 저장소에 반영)
- 개발자는 컴포넌트 or 디렉토리를 프로젝트 저장소(공유 저장소)에서 자신의 개인 작업공간(PWS)으로 체크아웃
- 자신의 개인공간에서 이 복사본에 대해 작업을 수행하고 끝나면 컴포넌트를 체크인하여 저장소로 돌려준다.
- 만약 두명이상이 동시에 한 컴포넌트에 대해 작업하려면, 각각은 저장소로부터 컴포넌트에 대해 체크아웃
  > 만약 어떤 컴포넌트가 체크아웃했으면, 체크아웃을 원하는 다른 사용자에게 이미 체크아웃된것을 알림
  > 시스템은 수정된 컴포넌트가 체크인 시, 다른 버전들은 다른 버전 식별자를 할당받고 따로 저장된 것을 보장
- 그림 25.5



6. 분산 버전 제어 Distributed version control

- 마스터 저장소가 개발팀에 의해 생산되는 코드를 유지하는 서버에 생성된다.
- 개발자는 체크아웃대신 프로젝트 저장소의 복제본을 생성하여 자신의 컴퓨터에 설치
- 자신의 컴퓨터에 설치된 개인 저장소에서 필요한 파일 작업을 수행하고 새로운 버전 유지
- 변경 작업 완료시, commit을 통해 이 변경이 반영되어 개인 저장소가 변경된다.
- 이 변경들이 프로젝트 저장소에 반영되도록 push하기
- 통합 관리자에게 변경된 버전을 이용할 수 있다고 알리기
- 변경된 내용을 프로젝트 저장소로부터 가져올 땐 Pull하기



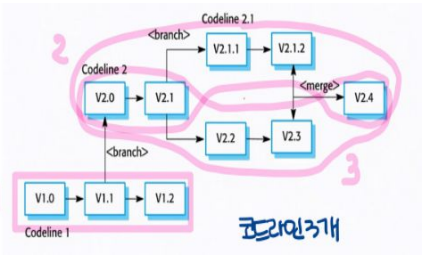
7. 분산 버전 제어 장점

1. 저장소에 대한 백업 기법을 제공
2. 같은 조직 내에서 작업하지 않고 여러 사람들이 같은 시스템에 동시에 작업할수 있는 오픈 소스 개발에 필수적

8. 저장소와 commit 구분

- 저장소와 관련 : pull, push

- commit 관련 : check in, check out



9. 분기와 병합

1. 분기 branch

  - 동일한 컴포넌트에 대한 독립적인 개발을 위해 코드라인의 분기가 필요하다

  - 다양한 목적으로 분기가 일어날 수 있다.

    여러 개발자들이 소스코드의 다른 버전들에 대해 독립적으로 작업시 그것들을 다른방식으로 수정할때

    새로운 분기가 만들어져야 하는 시스템상에서 작업을 할 때

    변경들이 동작중인 시스템을 우연히 고장 내지 않도록 하기 위해서 추천

2. 병합/합병 merge

  - 수행된 모든 변경사항들을 포함하는 컴포넌트의 새로운 버전을 만들기 위해 코드라인 분기들을 합병한다.

  - 서로 다른 부분을 수정했다면, 자동 병합이 가능

  - 변경 작업이 겹치는 경우, 수동 병합이 필요하며 충돌을 해결해야 함

10. 저장 공간 관리

- 버전 관리 시스템이 처음 개발되었을 때, 저장소 관리는 가장 중요한 기능 중의 하나였다. 디스크 공간의 값이 비싸서, 컴포넌트의 여러 복사본들에 의해 사용되는 디스크 공간을 최소화하는 것이 중요했다. 각각의 버전의 완전한 내용을 유지하는 것 대신에, 시스템은 버전 사이의 차이들의 목록을 저장하기로 했다.

 > 버전 간의 차이 delta를 이용하여 컴포넌트 버전을 저장

 > 각각의 완전한 버전을 저장하는 것보단 저장공간을 줄일 수 있다.

- 새 버전이 생성되면, 시스템은 사용되는 새 버전과 옛 버전의 차이인 델타를 저장한다.

  델타는 보통 변경된 라인들의 목록으로 저장되며, 이것들을 자동으로 적용하여

  컴포넌트의 한 버전은 다른 것으로부터 생성될 수 있다. 우린 후향 델타를 가장 많이 사용

- 델타의 문제점 : 모든 델타를 적용하는데 시간이 걸린다는 문제가 있다

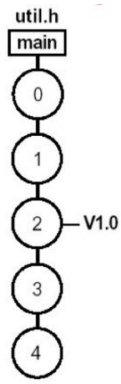
  > 이제 디스크 저장소는 상대적으로 값이 싸고 git은 다른 방식의 좀 더 빠른 접근법을 사용한다.

  > git은 델타를 사용하지 않음 -> 팩파일을 사용

11. 델타 종류 (delta : 버전간의 차이)

- 전향 델타 forward detla : 가장 초기 버전을 완전히 저장하고 이후 버전을 생성하기 위한 델타를 관리
- 후향 델타 backward delta : 가장 최근 버전은 완전히 저장하고 이전 버전을 생성하기 위한 델타를 관리

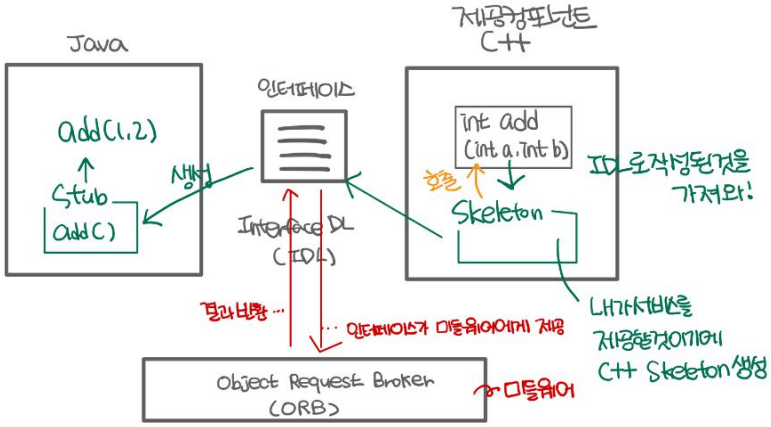
12. 추가 자료) Tags



- 태그는 label이라고도 한다.
- 이름을 부여하는 것이다.
- 요소의 모든 버전에 첨부하여 해당 버전을 쉽게 식별할 때 사용한다.
- 태그 이용방법 : V1.0 버전을 찾을때, 태그 부분을 보고 찾을 수 있다.
- 하나의 컴포넌트에 V1.0을 붙이는게 큰 의미를 부여하는게 아니라 이름을 부여하는거야 내 맘대로

Q. 나 Java는 add구현이 필요한데 이를 직접구현하지 않고 분산 컴포넌트에게 맡길래

그래서 직접호출이 불가능하니까 간접호출할 수 있도록 java는 stub을 만들어!



절차적 통신은

로컬 컴포넌트가 필요한 메서드를 직접 구현하지 않고 다른 컴포넌트의 메서드를 간접호출하여 이용하고 싶을때 분산 컴포넌트로 원격 프로시저 호출을 통해 구현하는 것입니다. 즉, 하나의 컴포넌트는 다른 컴포넌트가 제공하는 서비스를 로컬 프로시저나 메서드처럼 호출할 수 있습니다. 이때 미들웨어는 그들 사이의 통신을 도와주는 전달, 반환 역할을 해줍니다.

절차적 통신의 일반적인 통신 순서는 로컬 컴포넌트는 스텁을 호출하고 매개변수들을 전송 표준 표현으로 변환하여 미들웨어를 통해 원격 프로시저에 요청을 보내고 원격 프로시저는 매개변수들을 필요한 형식으로 변환하여 계산을 수행하고 미들웨어를 통해 결과를 반환한 뒤 로컬 컴포넌트는 스텁을 통해 반환값을 받아 사용합니다.

예시로 만약 로컬 컴포넌트의 언어가 Java이고 다른 컴포넌트의 언어가 c++이라고 가정하였을때 이 컴포넌트의 add 메서드를 로컬 컴포넌트가 간접호출로 불러와 사용하고 싶다고 생각해보겠습니다. 우선 원격 프로시저 호출인 RPC는 로컬 컴포넌트에서 접근할 수 있도록 호출될 프로세스를 위한 스텁을 요구합니다. 그러면 로컬 컴포넌트는 java 스텁을 호출하고 매개변수들을 전송 표준 표현으로 변환하여 미들웨어를 통해 원격 프로시저에 요청을 보냅니다. 제공 컴포넌트는 c++ skeleton을 생성하여 IDL로 작성된 것을 가져오라고 하여 매개변수들을 필요한 형식으로 변환하여 계산을 수행하고 이를 미들웨어를 통해 결과를 반환하니다. 로컬 컴포넌트는 스텁을 통해 반환값을 받은 add를 간접 호출하여 이용할 수 있게 됩니다.