오픈소스소프트웨어

Open-Source Software

ICT융합학부 조용우

SW개발 생명주기

SW개발 생명주기

- 의미
  - SW를 어떻게 개발할 것인가에 대한 추상적 표현
  - 순차적 또는 병렬적 단계로 구성됨
  - 개발 모델 또는 SW공학 패러다임이라고도 함

- 특징
  - 개발 생명주기의 각 단계에 관련된 활동들이 정의되어 있음
  - 단계별 활동들을 통해 다음 단계에 활용될 수 있는 산출물이 작성됨
  - 전체 프로젝트의 비용 산정과 개발 계획을 수립할 수 있는 기본 골격 제시
  - 참여자들 간에 의사소통의 기준과 용어의 표준화를 가능하게 함
  - 문서화가 충실한 프로젝트 관리를 가능하게 함

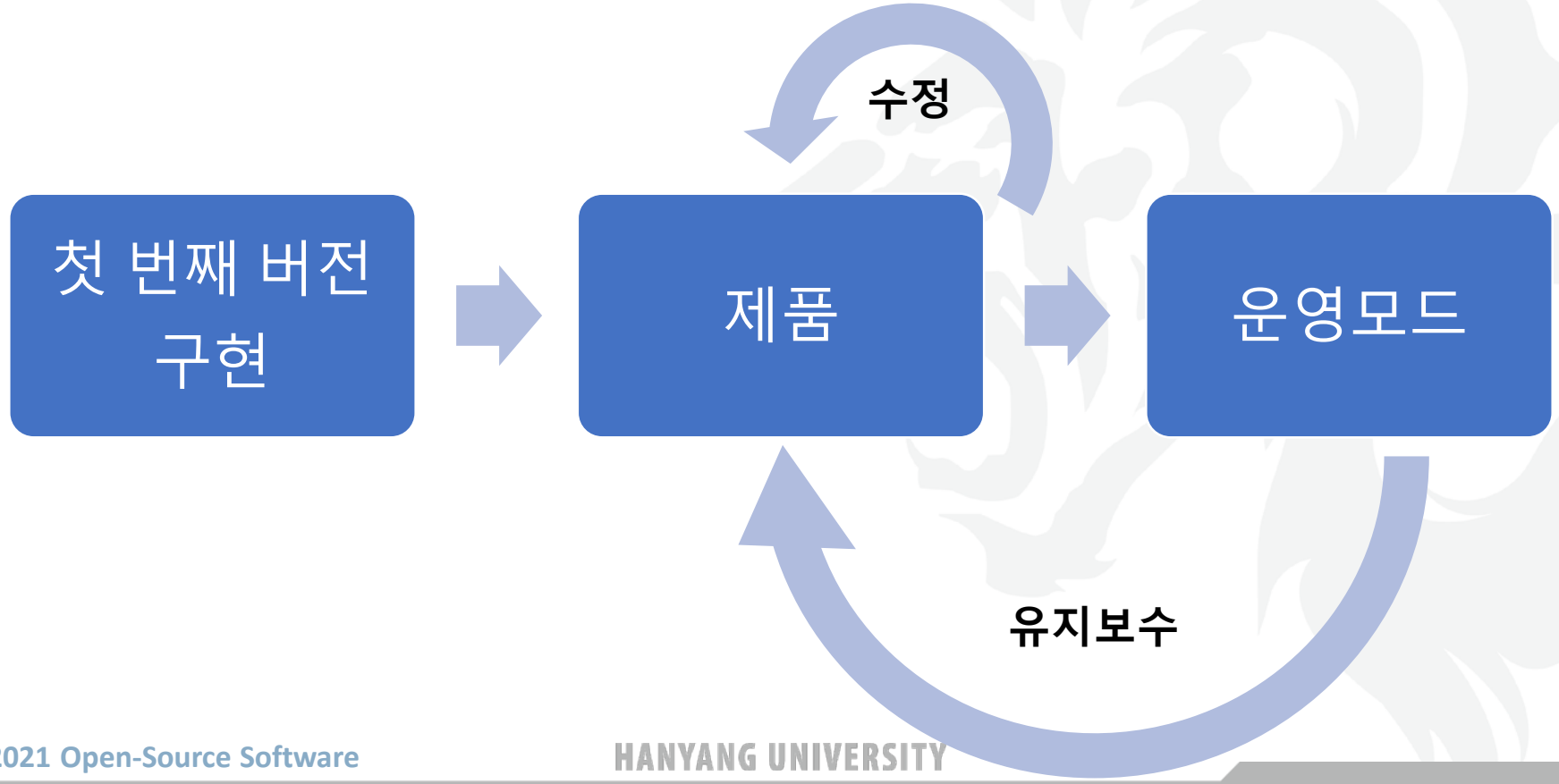
SW개발 생명주기 모델의 종류

- 주먹구구식 개발 모델(Build-Fix Model)
- 폭포수 모델(Waterfall Model)
- 원형 모델(Prototyping Model)
- 나선형 모델(Spiral Model)

주먹구구식 개발 모델(Build-Fix Model)

- 개요
  - → 요구사항 분석, 설계 단계 없이 일단 개발에 들어간 후 만족할 때까지 수정작업 수행
- 적용 가능한 경우
  - → 크기가 매우 작은 규모의 SW개발
- 단점
  - → 정해진 개발 순서가 없기 때문에
    - 계획이 정확하지 않음
    - 관리자는 프로젝트 진행 상황 파악에 어려움
    - 개발 문서가 없기 때문에 개발 및 유지보수에 어려움
    - 이후 체계적인 SW개발 생명주기 모델의 연구를 가져옴

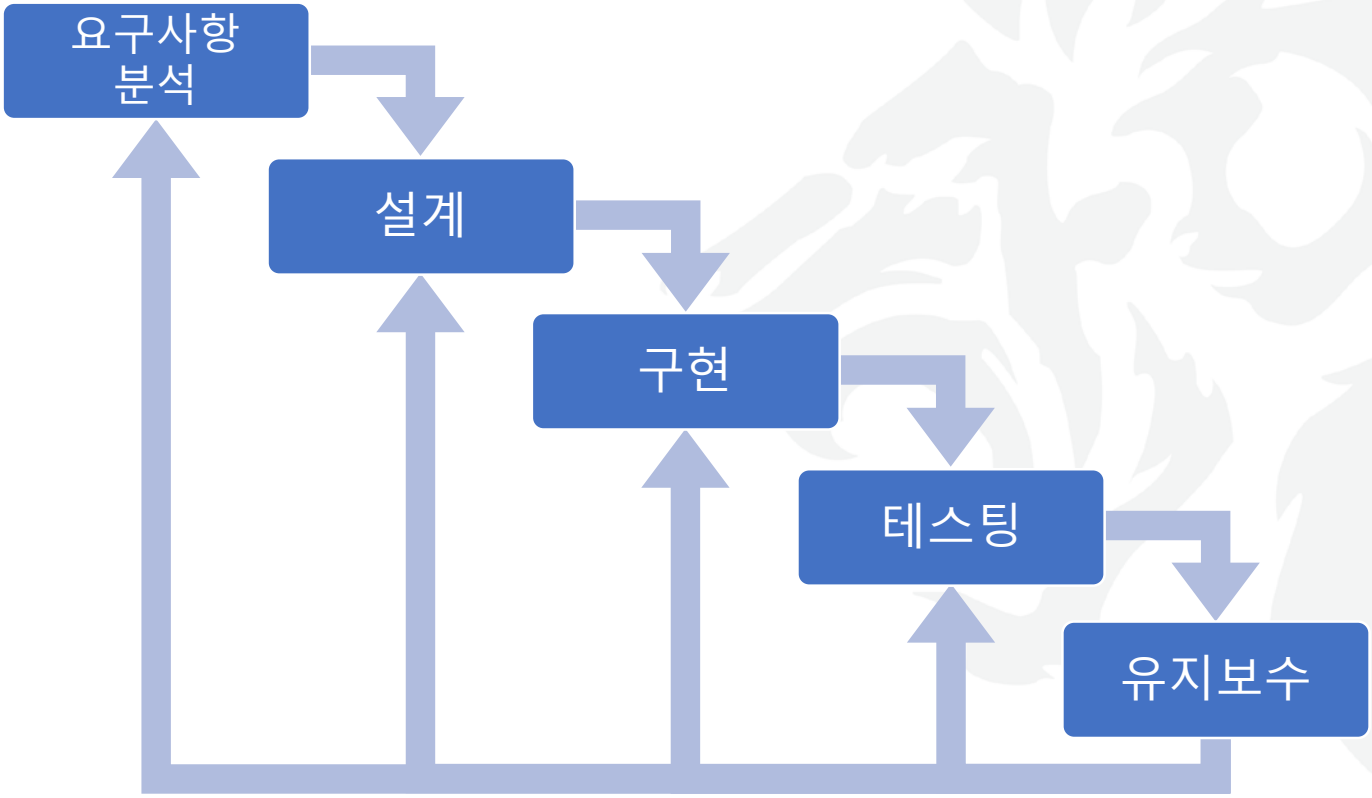
주먹구구식 개발 모델(Build-Fix Model)



폭포수 모델(Waterfall Model)

- 개요
  - 순차적으로 SW를 개발하는 전형적인 개발 모델
  - 대부분의 SW개발 프로젝트의 기본적 모델이며 가장 많이 사용되는 모델
  - SW개발의 전 과정을 나누어 체계적이고 순차적으로 접근하는 방법
    - 개발 과정: 요구사항 분석, 설계, 구현, 테스팅, 유지보수

폭포수 모델(Waterfall Model)



폭포수 모델(Waterfall Model)

- 단계별 활동
  - 요구사항 분석
    - 개발하고자 하는 SW에 대한 요구사항 수집, 문제 이해 및 분석 단계
    - SW엔지니어 또는 분석가: 고객의 요구사항을 기능, 성능, 인터페이스 등으로 파악하고 문서화
    - 산출물: 요구사항 명세서(Requirement Specification)
  - 설계
    - 프로그램의 데이터 구조, SW구조, 인터페이스 구조, 알고리즘 등 모든 시스템의 구조 결정
    - 산출물: 설계 명세서
  - 구현
    - 설계 명세서를 시스템의 실제 모습으로 변환 시키는 것
    - 산출물: 소스 코드 및 프로그램
  - 테스트
    - 프로그램이 입력에 따라 요구되는 결과대로 작동하는지, 내부적 이상 여부 및 오류 발견을 위해 수행
    - 테스트 계획을 세운 후 문서화
  - 유지보수
    - 개발된 SW의 변경사항을 수정하는 것
    - 수정 유지보수, 적응 유지보수, 기능 추가 유지보수 등이 있음

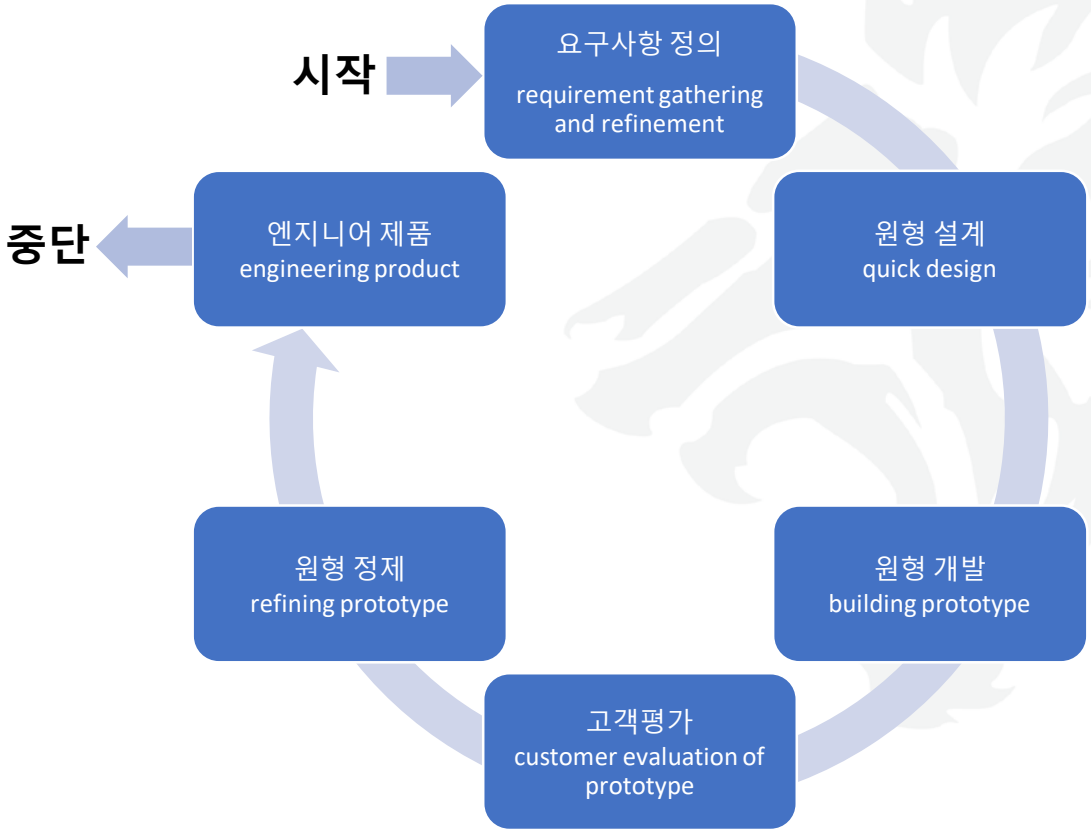
폭포수 모델(Waterfall Model)

- 장점
  - 각 단계별로 정형화된 접근 방법 가능
  - 체계적인 문서화가 가능하여 프로젝트 진행을 명확하게 할 수 있음
- 단점
  - 앞 단계가 완료될 때까지 다음 단계들은 대기 상태여야 함
  - 실제 작동되는 시스템을 개발 후반부에 확인 가능하기 때문에 고객이 요구사항 확인하는데 많은 시간이 걸림

원형 모델(Prototyping Model)

- 개요
  - 폭포수 모델의 단점을 보완한 모델
  - 점진적으로 시스템을 개발해 나가는 접근 방법
  - 원형(Prototype)을 만들어 고객과 사용자가 함께 평가한 후 개발될 SW의 요구사항을 정제하여 보다 완전한 요구사항 명세서를 완성함
- 목적
  - 원형을 가능한 빨리 개발하여 고객과 검증하는 것
  - 방법
    - 고객으로부터 피드백을 받은 후 원형을 폐기
    - 시스템 기능 중 중요한 부분만 구현하여 피드백을 얻은 후 지속적으로 발전시켜 완제품을 제작
- 적용 가능한 경우
  - SW개발 초기에 고객 요구사항을 완전히 파악하기 어려울 때

원형 모델(Prototyping Model)



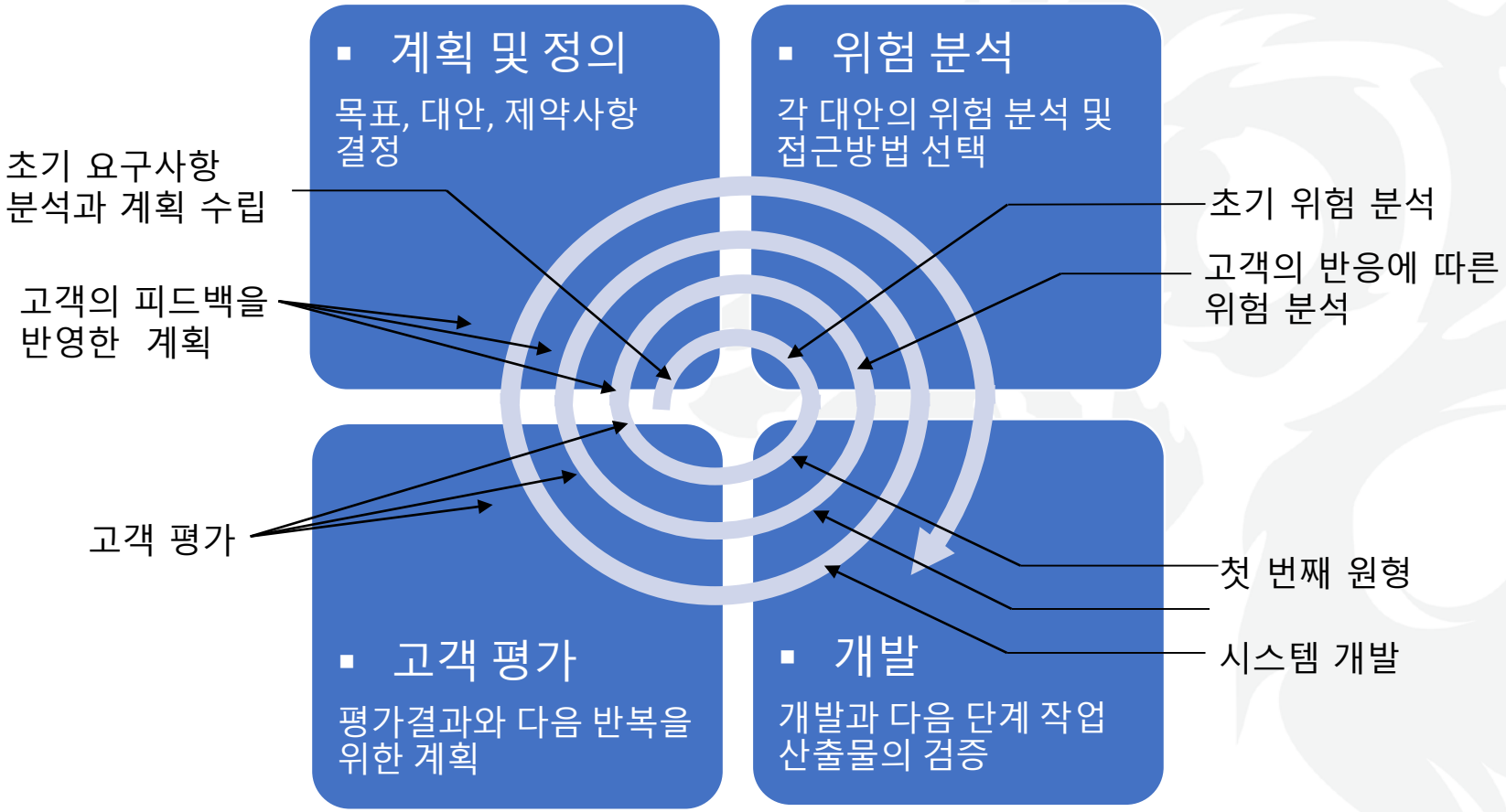
원형 모델(Prototyping Model)

- 단계별 활동
  - 요구사항 정의
    - 고객의 일부 요구사항 또는 불완전한 요구 사항으로부터 제품의 윤곽을 잡음
  - 원형 설계
    - 주어진 요구사항을 기반으로 빠른 설계를 함
    - 주로 제품의 사용자 인터페이스에 초점을 맞춤
  - 원형 개발
    - 설계된 원형을 RAD(Rapid Application Development) 도구 등을 사용하여 빠르게 구현함
    - 고객이 요구하는 기능을 구현하고 필요한 요소를 파악하는데 중점을 둠
    - 프로그램의 신뢰도나 품질이 아니라 가능한 빨리 원형을 구현하는 것이 목적
  - 고객 평가
    - 고객과 개발자가 함께하는 가장 중요한 단계
    - 고객 요구사항을 정확하게 규명하기 위해 원형에 대한 사용 및 평가 시간을 충분히 제공
    - 개발될 SW의 요구사항 정제에 중요한 정보로 활용
  - 원형 정제
    - 원형이 어떻게 수정되어야 할지를 결정함
    - 원형 개발과 검증, 요구사항 정제의 순환을 반복하여 추가적인 정보를 통해 요구사항을 완성해 나감

나선형 모델(Spiral Model)

- 개요
  - 폭포수 모형과 원형 모형의 장점을 수용하고 위험 분석(Risk analysis)을 추가한 점증적 개발 모델
  - 프로젝트 수행 시 발생하는 위험을 관리하고 최소화 하려는 것이 목적
- 특징
  - 여러 개의 작업 영역으로 구분
  - 나선상의 각 원은 SW개발의 점증적 주기 표현
    - 가장 안쪽 타원부터 개념적 개발 프로젝트, 실제 제품 개발 프로젝트, 제품 향상 프로젝트, 유지보수 프로젝트
  - 단계가 명확히 구분되지 않고, 엔지니어가 프로젝트 성격이나 진행 상황에 따라 단계 구분

나선형 모델(Spiral Model)



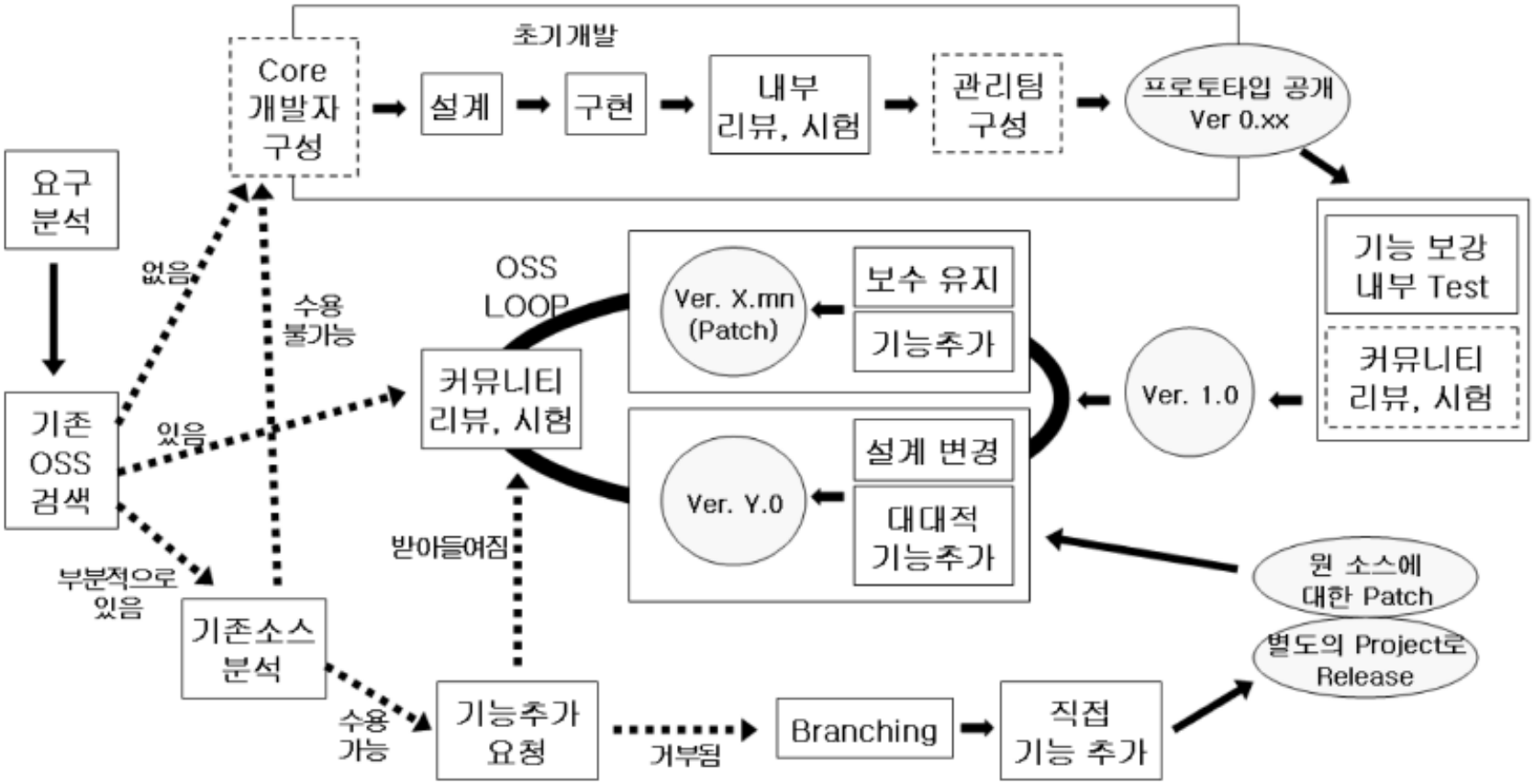
나선형 모델(Spiral Model)

- 단계별 활동
  - → 계획 및 정의 단계
    - 개발자는 고객으로부터 요구사항을 수집
    - 개발자는 시스템의 성능, 기능을 비롯한 시스템의 목표를 규명하고 제약 조건을 파악
    - 목표와 제약 조건에 대한 여러 대안들을 고려하고 평가함으로써 프로젝트 위험의 원인을 규명 가능
  - → 위험 분석 단계
    - 초기의 요구 사항을 토대로 위험 규명
    - 위험에 대한 평가가 이루어지면 프로젝트를 계속 진행할 것인지 아니면 중단할 것인지를 결정
  - → 개발 단계
    - 시스템에 대한 생명주기 모델을 선택하거나 원형 또는 최종적인 제품을 만드는 단계
  - → 고객 평가 단계
    - 구현된 SW(시뮬레이션 모형, 원형 또는 실제 시스템)를 고객이나 사용자가 평가함
    - 고객의 피드백을 얻는데 필요한 작업이 포함
    - 다음 단계에서 고객의 평가를 반영할 수 있는 자료 획득 가능

나선형 모델(Spiral Model)

- 적용 가능한 경우
  - → 개발에 따른 위험을 잘 파악하여 대처할 수 있기 때문에
    - 고비용의 시스템 개발
    - 시간이 많이 소요되는 큰 시스템 구축 시 유용
- 장점
  - → 프로젝트의 모든 단계에서 기술적인 위험을 직접 고려할 수 있어 사전에 위험 감소 가능
  - → 테스트 비용이나 제품 개발 지연 등의 문제 해결 가능
- 단점
  - → 개발자가 정확하지 않은 위험 분석을 했을 경우 심각한 문제 발생 가능
  - → 폭포수, 원형 모델에 비해 상대적으로 복잡하여 프로젝트 관리 자체가 어려울 수 있음

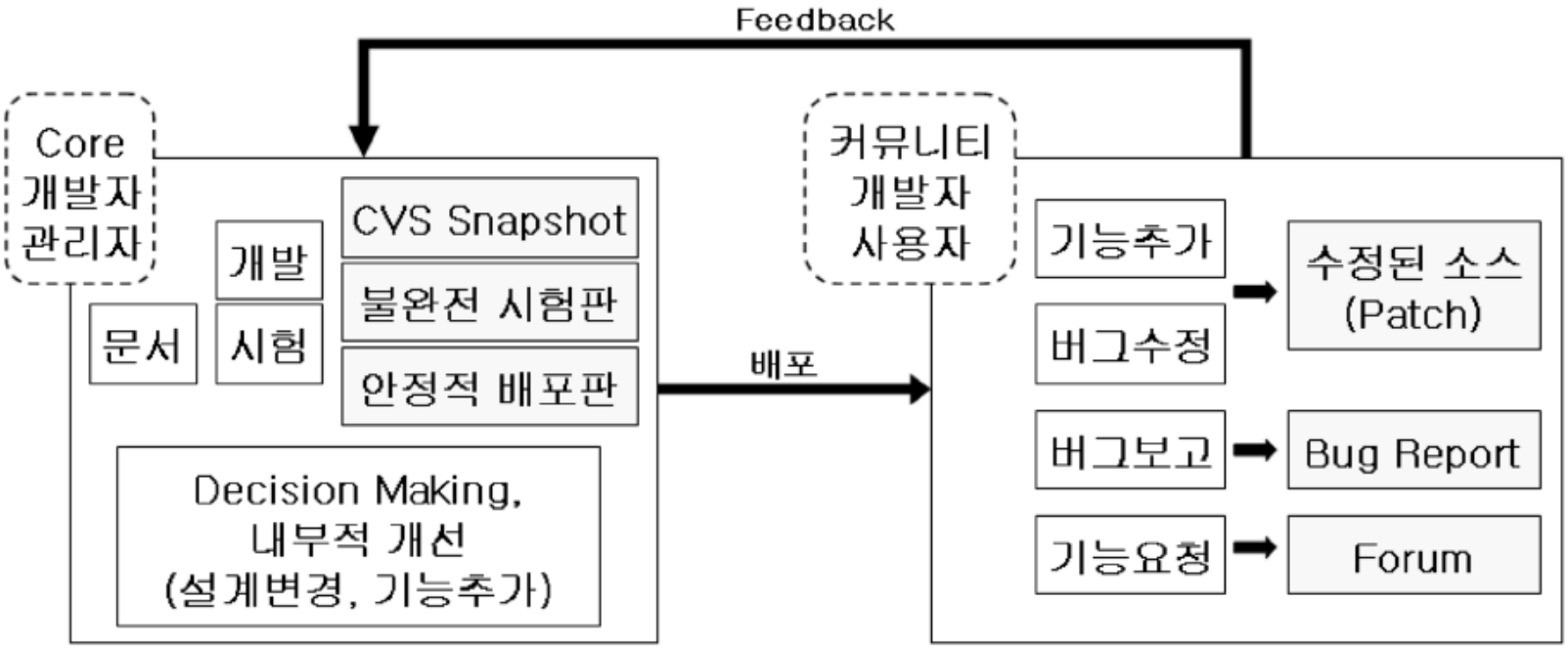
오픈소스SW의 생명주기



오픈소스SW 개발단계

- 새로운 프로젝트 만들기
- 프로토타입의 구현
- 결과물 배포
- 개발자간 소통
- 커뮤니티 기반 개발

오픈소스SW의 순환 구조



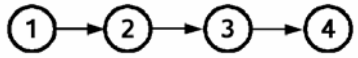
Version Control System

VCS (Version Control System)

- 버전 관리 시스템
  - → 동일한 정보에 대한 여러 버전을 관리하는 도구
    - 과거 및 현재 상태를 모두 저장
    - 상태 변화에 따른 모든 중간 단계 저장
  - → 팀 단위로 개발 중인 소스 코드 등의 디지털 문서의 버전을 관리
  - → 시간에 따른 변경사항과 그 변경사항을 작성한 작업자를 추적 관리

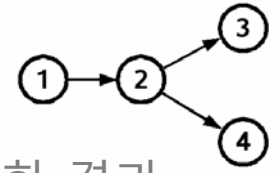
VCS 관련 용어

- 버전 번호(version number)



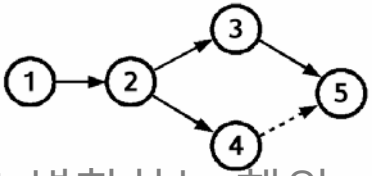
  - → 소스가 변경될 때마다 붙는 고유번호 (표지)

- 브랜치(branching / branch)

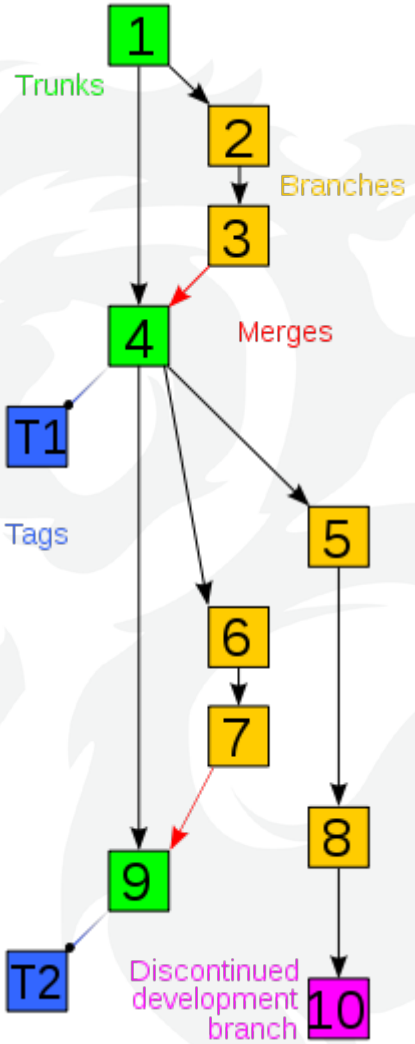


  - → 소스를 중간에 분기하는 행위 / 분기한 결과

- 머지(merge)

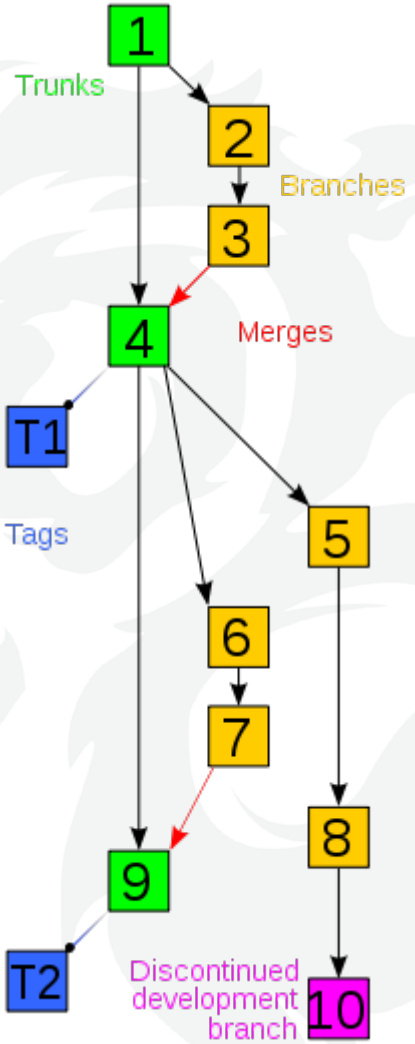


  - → 두 개의 분기된 브랜치를 하나로 병합하는 행위



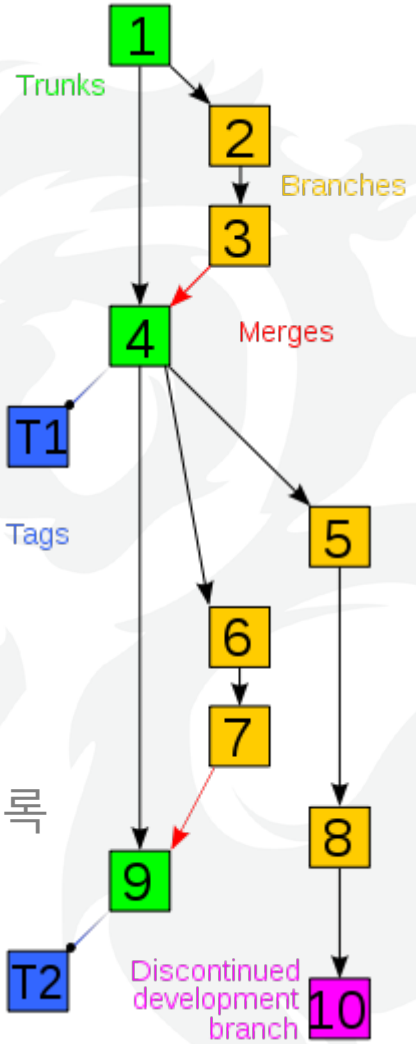
VCS 관련 용어

- 본류(trunk)
  - 태그가 붙지않은 이름없는 브랜치
  - 프로젝트에서 가장 중심이 되는 디렉토리
  - 모든 개발작업은 trunk디랙토리에서 이뤄짐
- 리비전(revision)
  - 새롭게 변경된 결과물
  - 특정시간/특정 브랜치의 어떤 한 상태
  - 상태에 해당하는 인덱스를 리비전 번호 (revision number)라고 함
- 태그(tag)
  - 특정한 리비전을 나중에 찾거나 알아보기 위해 붙인 텍스트



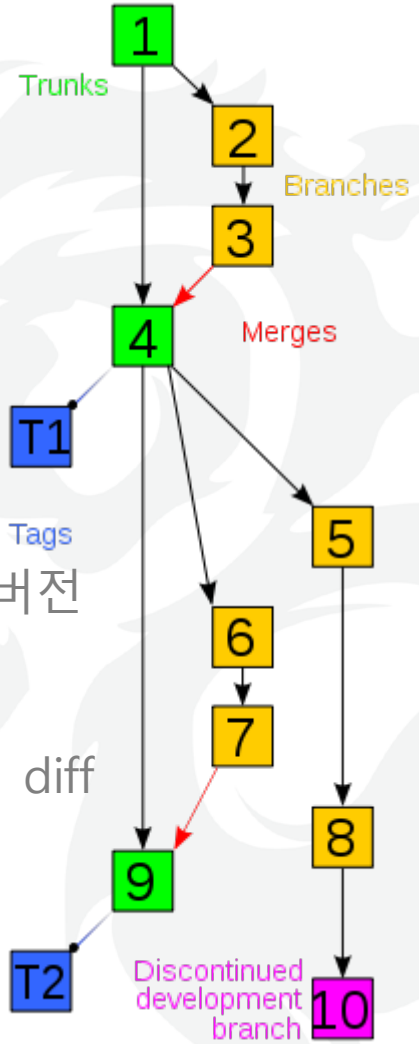
VCS 관련 용어

- 저장소(repository)
  - → 소스 (코드, 문서, 그림 등)을 저장하는 장소
- 커밋(commit)
  - → 저장소에 변경 사항을 반영하는 행위
- 커밋 로그(commit log)
  - → 커밋할 때 해당 커밋이 어떠한 변경을 했는 지 작성한 기록



VCS 관련 용어

- 체크 아웃(check out)
  - → 저장소에서 파일을 가져옴
- 체크 인(check in)
  - → 체크 아웃한 파일의 수정이 끝난 경우 저장소에 새로운 버전으로 갱신하는 일
  - → 이전에 갱신한 것이 있을 경우, 충돌(conflict)을 알려주며 diff 도구를 이용해 수정하고 커밋하는 과정을 거치게 됨



왜 VCS를 사용하나?

- 무언가 잘못되었을 때 복구를 돕기 위하여
- 프로젝트 진행 중 과거의 어떤 시점으로 돌아갈 수 있게 하기 위하여
- 여러사람이 같은 프로젝트에 참여할 경우, 각자가 수정한 부분을 팀원 전체가 동기화하는 과정을 자동화하기 위하여
- 소스 코드의 변경 사항을 추적하기 위하여
- 소스 코드에서 누가 수정했는지 추적하기 위하여

왜 버전 관리 시스템을 사용하나?

- 대규모 수정 작업을 더욱 안전하게 진행하기 위하여
- 브랜치(Branch)로 프로젝트에 영향을 최소화 하면서 새로운 부분을 개발하기 위하여
- 머지(Merge)로 검증이 끝난 후 새로이 개발된 부분을 본류(trunk)에 합치기 위하여
- 많은 오픈 소스 프로젝트에서 어떠한 형태로든 버전 관리를 사용하고 있으므로
- 코드의 특정 부분이 왜 그렇게 쓰여 졌는지 의미를 추적하기 위하여

어떻게 VCS를 사용하나?

1. 갑돌이가 어떤 파일을 저장소(repository)에 추가(add)한다.
2. 추가되었던 파일을 갑돌이가 체크 아웃(check out) 한다.
3. 갑돌이가 인출된 파일을 수정한 다음, 저장소에 커밋(commit) 하면서 설명을 붙인다.
4. 다음날 을순이가 자신의 작업 공간을 동기화(update) 한다. 이때 갑돌이가 추가했던 파일이 전달된다.
5. 을순이가 추가된 파일의 커밋 로그(commit log)을 보면서 갑돌이가 처음 추가한 파일과 이후 변경된 파일의 차이를 본다(diff).

중앙집중식 VCS

- 특징
  - 중심이 되는 저장소가 존재
  - 메인 트리가 존재함
  - 저장소 의존도가 크므로 브랜칭을 쉽게 하지 않음
    - 중요한 기능단위/개발단위로만 브랜칭
    - 의존성 문제로 한 번 만든 브랜치는 쉽게 없애지 않음
  - 연속된 숫자를 인덱스로 사용
    - 예) 3125
- (예) CVS, Subversion

분산식 VCS

- 특징
  - → 중심이 되는 저장소가 존재하지 않음
  - → 브랜칭과 머징이 매우 자유로움
    - 새로운 코드를 만들거나 수정할 때 브랜칭을 하는 것이 일반적임
  - → 고유의 리비전 인덱스를 사용
- (예) Mercurial (hg), git

Issue Tracker (Bug Tracker)

- 이슈 트래커 (버그 트래커)
  - → 제품의 문제점을 발견, 해결하는 과정을 시스템화
  - → 문제 발견, 증상 규격화, 원인규명, 재구현, 문제 해결의 과정을 거침
  - → VCS와 연동하여 VCS에 관련된 다양한 작업을 지원
  - → 작업 분담 / 이슈 관리 / 변경 이력 추적 / 문서화
  - → (예) Bugzilla, Trac, Jira, Redmine

CI (Continuous Integration)

- 지속적 통합 도구
  - → 지속적으로 퀄리티 컨트롤을 적용하는 프로세스를 실행
  - → cf)고전적인 방법: 모든 개발 완료 후 퀄리티 컨트롤
  - → 작은 단위의 작업, 빈번한 작업
  - → 현재 코드를 복사한 후, 변경하고, 제출하면, 그 사이 나머지 내용의 변경 사항이 발생함 → 자주 통합해서 한 번 통합할 때 복잡함을 줄임

CI (Continuous Integration)

- 지속적 통합 도구
  - → 코드의 변경 내용을 확인하고 서비스를 계속 최신 상태로 유지
  - → 최근 리비전의 무결성을 확인
    - 코드 오류 검사
      - » 단위 테스트 (unit test) / 기능 테스트 (functional test)
    - 코드의 호환성 검사
  - → 무결성 결과 보고
  - → (예) Jenkins, TeamCity, GitLab CI, CircleCI, Travis CI, Codeship

문서 및 지역화 도구

- 문서 및 지역화 도구
  - → 프로그램 소개 및 활용 범위 (readme)
  - → 디렉토리/파일 구조도 및 설명 (files)
  - → 프로그램 설치 방법 (install)
  - → SW 옵션 및 사용법 (usage)
  - → 운영에 대한 FAQ (faq)
  - → SW 변경 사항 (changes)
  - → 소스 코드 라이선스 표시 (license)
  - → (예) Doxygen, Wiki, Gettext, JavaDoc

Hosting Services

- 오픈 소스 프로젝트 호스팅 서비스
  - → 여러 곳의 개발자들의 자발적 참여를 지원해야 함
  - → 운영에 필요한 다양한 소통 수단과 개발 도구를 총괄적으로 모아서 서비스로 제공
  - → (예) SourceForge.net, KLDP.net, Google Code, etc.