AI 프로그래밍(SW개발)

융합학과 권오영
oykwon@koreatech.ac.kr

Grady Booch, “The History of Software Engineering,” ComputingEdge, Sep. 2019, pp. 8 ~ 14

소프트웨어 공학의 역사

❖ 용어의 기원

- Margaret Hamilton: 구전에 의하면 하드웨어 엔지니링과 구별하기 위해 1963 혹은 1964년에 처음 사용 (SAGE(Semi-automatic Ground Environment) 프로그램으로 Draper Lab. 에서 Skylab과 Apollo 개발을 선도)

❖ Software Engineering versus Computer Science

- Grace Hopper: **Programming** is a practical art.
- Edsger Dijkstra: The art of **programming** is the art of organizing complexity.
- Donald Knuth: **programming** as art because it produced objects of beauty.
- 소프트웨어공학은 기술(art)이자 과학이다. 실용적인 기술이다.
- 소프트웨어공학은 비용, 일정, 복잡성, 기능, 성능, 신뢰성 및 보안, 아울러 법적이며 윤리적인 행위들 간의 균형을 맞추어야 한다.

소프트웨어공학의 역사

❖ 19th 세기 ~ 20th 세기 초: Human Computers
- 사람들이 컴퓨터고, 데이터가 들어오면 컴퓨터가 계산을 하고 그 결과를 다음 컴퓨터로 전달하는 방식 (Pipeline)

❖ 대공황부터 2차 세계대전시기: 전자식 컴퓨터의 탄생
- 초기 범용 컴퓨터 개발 (stored program computer)
- 독일: Z2,Z3,Z4; 영국: Colossus; 미국: ENIAC, EDVAC

❖ 2차 세계대전 이후: 컴퓨팅의 부상과 소프트웨어공학의 탄생
- 순서도(Flowchart), 알고리즘 분해, 프로그래밍 언어의 등장
- IBM System/36 등장: 하드웨어로부터 소프트웨어 분리
  - ✓ 소프트웨어가 개별 경제적 가치를 갖는 요소로 개발

소프트웨어 공학

❖ 냉전의 시기: 성장기

- 미국과 소련의 경쟁
- 혁신이 발생
  - ✓ Human-computer interface (CRT display, light pen)
  - ✓ Core memory
  - ✓ 분산환경의 거대규모 소프트웨어 시스템
- 소프트웨어 개발이 고비용 부분이자 가장 중요한 부분으로 되어감
- Time Sharing 아이디어 (Christopher Strachey)의 발전
- The Mythical Man Month (Fred Brooks):
  지연이 발생한 소프트웨어 개발에 인력이 더 투입되더라도 communication 비용의 증가로 소프트웨어 개발일정에 도움이 되지 않는다.
  - ✓ 소프트웨어 공학은 기술적 프로세스일 뿐만 아니라 매우 인간적인 프로세스이기도 하다.
- 프로그래머는 컴퓨터보다 고비용이고, 컴퓨터를 어디에나 두는 것이 경제적이다.

소프트웨어 공학의 역사

❖ 60년대에서 80년대: 성숙기
- 여전히 냉전 시기가 지속
- 구조적 프로그래밍(Structured programming) 개념(Edsger Dijkstra) 등장
  - ✓ 순차, 조건(분기), 반복의 구조들의 결합으로 프로그래밍
- 정형 소프트웨어 개발 프로세스(Formal software development process)(Winston Royce)
  - ✓ 폭포수모델(waterfall process)
  - ✓ 반복개발(Iterative development), 프로토타이핑(prototyping), 소스코드를 넘어서는 가공물의 가치정립
- Entity-relationship modeling (Peter Chen)
- 구조화된 분석 및 설계; 소프트웨어 검사(software inspections); 정보공학(information engineering); 함수형 프로그래밍(functional programming); 분산컴퓨팅(distributed computing)
  - ✓ 최초의 황금기

❖ 80년대: 황금기
- 소프트웨어품질; 초대형 소프트웨어집중시스템; 소프트웨어의 세계화; 프로그램에서 분산시스템으로 전이
- 객체지향프로그래밍(Object-Oriented programming; Ole Dahl and Kristern Nygaard)
- 객체모델링(Object modeling)

소프트웨어 공학의 역사

- ❖ 90년대와 밀레니엄: 파괴의 시기 (새로운 혁신의 시기)
  - 인터넷의 등장
  - 요구분석, 설계, 개발, 테스팅, 그리고, 구성관리(configuration management)로 분화
  - 점진적이고 반복적인 개발을 통한 지속적인 통합 (CI/CD)
  - 오프소스 프레임워크 (Eric Raymond)
  - 요소기반 공학이 서비스기반아키텍쳐, 마이크로서비스아키텍쳐로 변형
  - 컴퓨팅이 메인프레임에서 데이터센터를 거쳐 클라우드로 이동
  - 더 이상 단순한 프로그램이나 모놀리식 시스템을 구축하지 않고, 가장자리(edge)에서 분산시스템과 상호작용하는 앱구축 (Edge compution)
  - 애자일방법 등장 (Agile method)
    - ✓ Scrum
    - ✓ Extreme programming
    - ✓ Refactoring
  - Git; Github; Stack Overflow; – 버전관리; SW개발에 대한 질의응답
  - Computational Thinking; DevOps; Full stack develop; IoT 개념의 등장
  - SWEBOK(Software Engineering Body of Knowledge); SEBOK(Systems Engineering Body of Knowledge) 정립

소프트웨어 공학의 역사

❖ 향후 10년: 빅데이터와 AI의 새로운 시즌

- 인공지능: 이미지, 비디오, 오디오 신호의 복잡한 패턴을 찾기 위해 신경망과 경사하강법을 사용
  - ✓ 거대규모 인공지능 시스템들의 등장
- 양자 컴퓨팅(Quantum Computing)

- 소프트웨어 공학의 근본
  - ✓ 건전한 추상화 형성 (Craft sound abstraction)
  - ✓ 관심사들에 대한 명확한 분리 (Maintain a clear separation of concerns)
  - ✓ 균형잡힌 책임과 분배를 위한 노력 (Strive for a balanced distribution of responsibilities)
  - ✓ 단순함의 추구 (Seek simplicity)

Software is the **invisible writing**
that whispers the stories of possibility to our hardware.
And you are the **storytellers**. – Grady Booch

소프트웨어 개발 과정

(참고: 김원외 9, 실용적 컴퓨팅 사고와 소프트웨어, 생능출판사, 2018)

컴퓨터 소프트웨어

❖ Programming Step (Problem solving step)

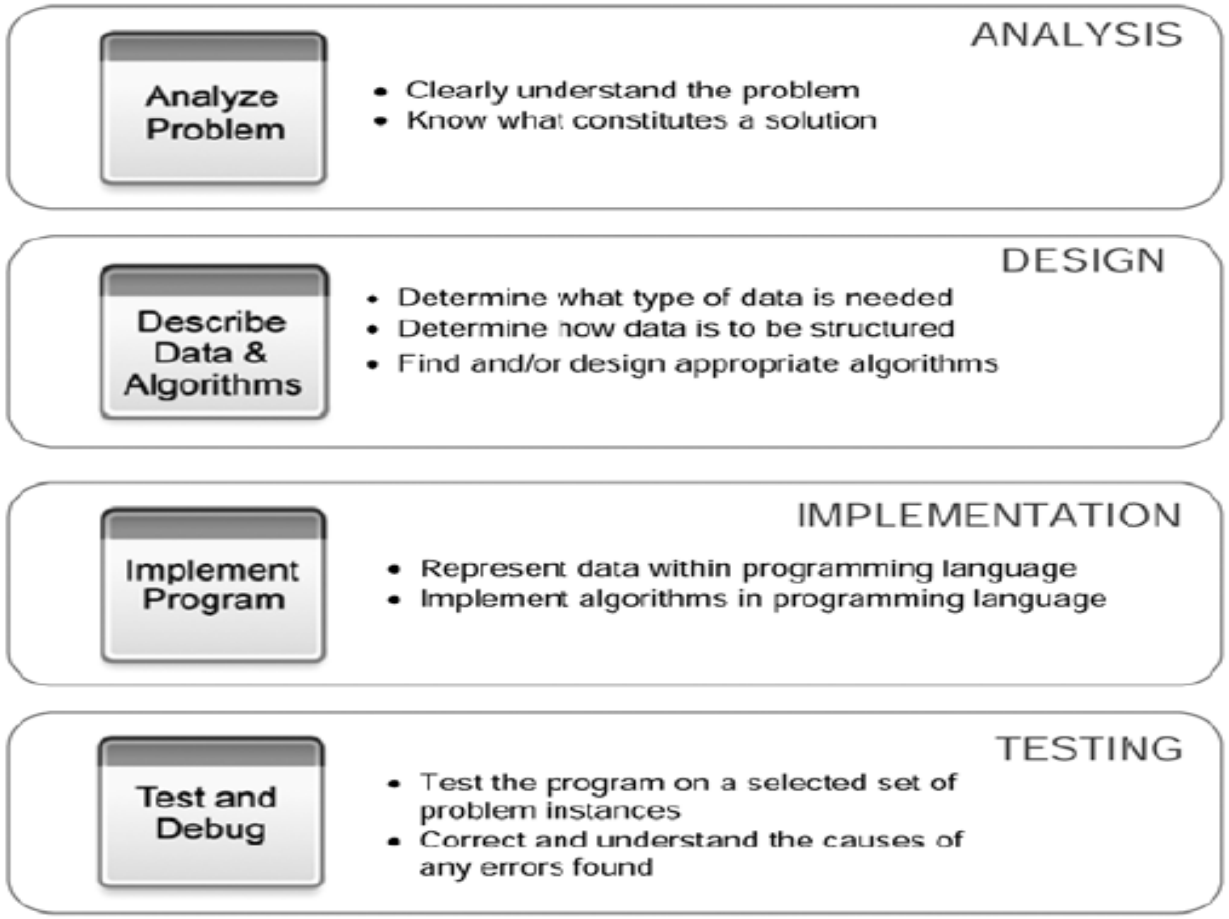
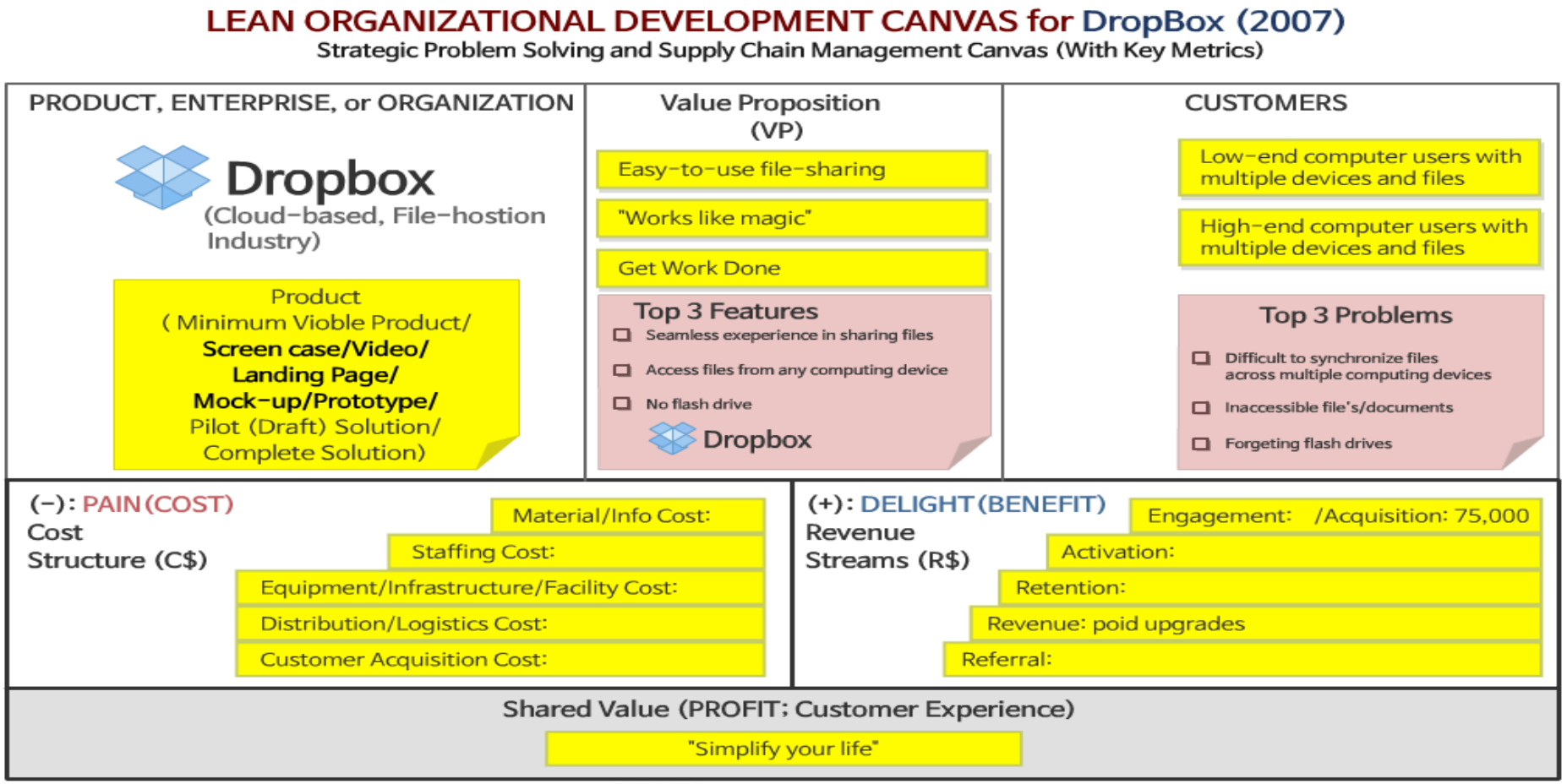


FIGURE 1-22 Process of Computational Problem Solving

소프트웨어 개발

- ❖ 규모가 큰 소프트웨어 개발
- ❖ Lean Canvas (사업계획서)



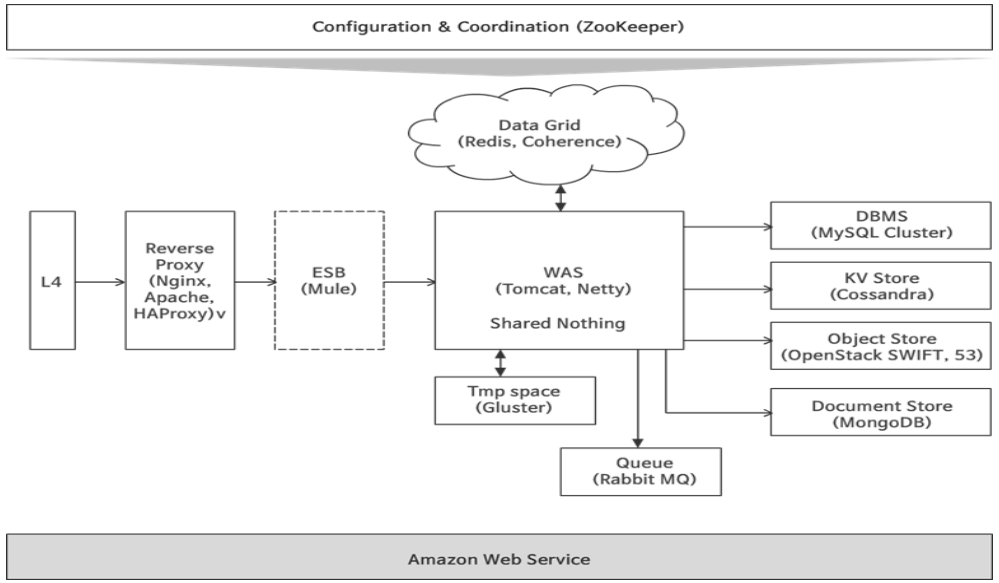
KOREATECH 한국기술교육대학교

소프트웨어 개발

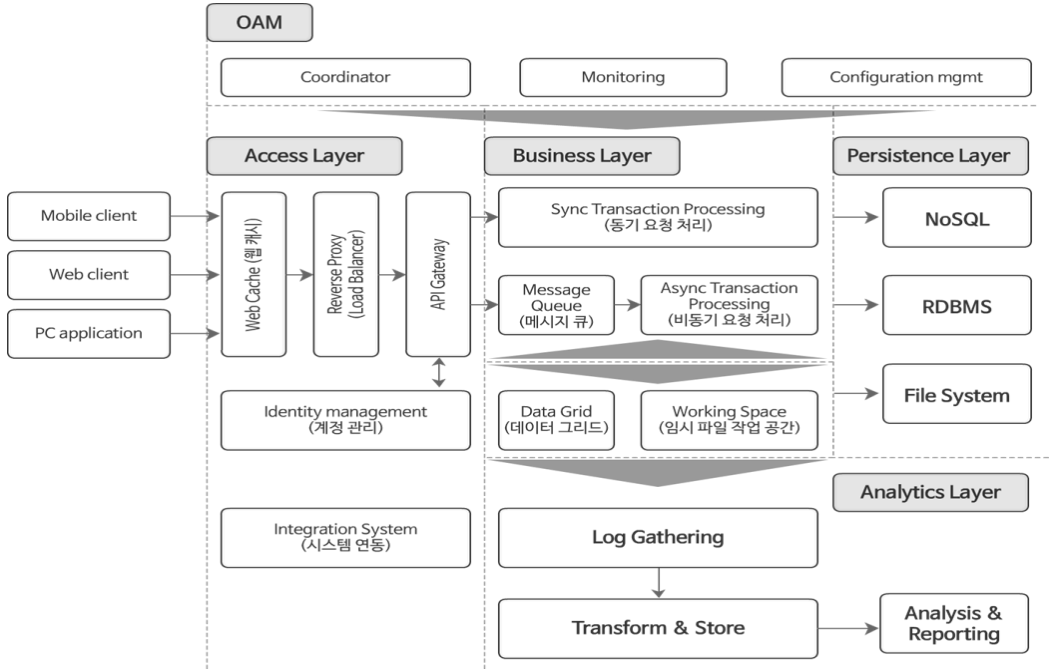
❖ 규모가 큰 소프트웨어 개발

- 추상화에 의한 계층화 및 모듈화

아마존 웹 서비스



참조 아키텍쳐

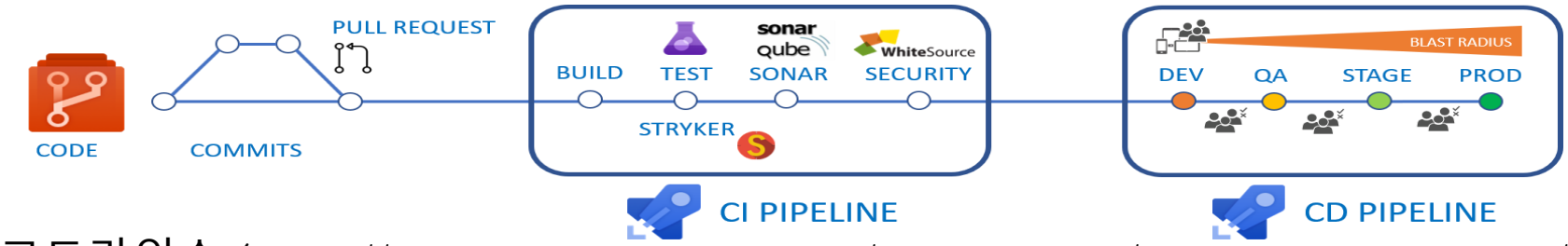


KOREATECH 한국기술교육대학교

소프트웨어 개발

❖ 대규모 소프트웨어 개발

- 계층별 모듈별 개발
- 협업 (개발자와 개발자, 개발자와 시험자, 개발자와 운영자, 개발자와 고객)
- 개발순서 및 관리가 필요
  (https://opensource.com/article/19/7/cicd-pipeline-rule-them-all)



- 코드라인수 (https://informationisbeautiful.net/visualizations/million-lines-of-code/)
  - ✓ Windows 7: 40,000,000; Facebook: 61,000,000; car software: 100,000,000

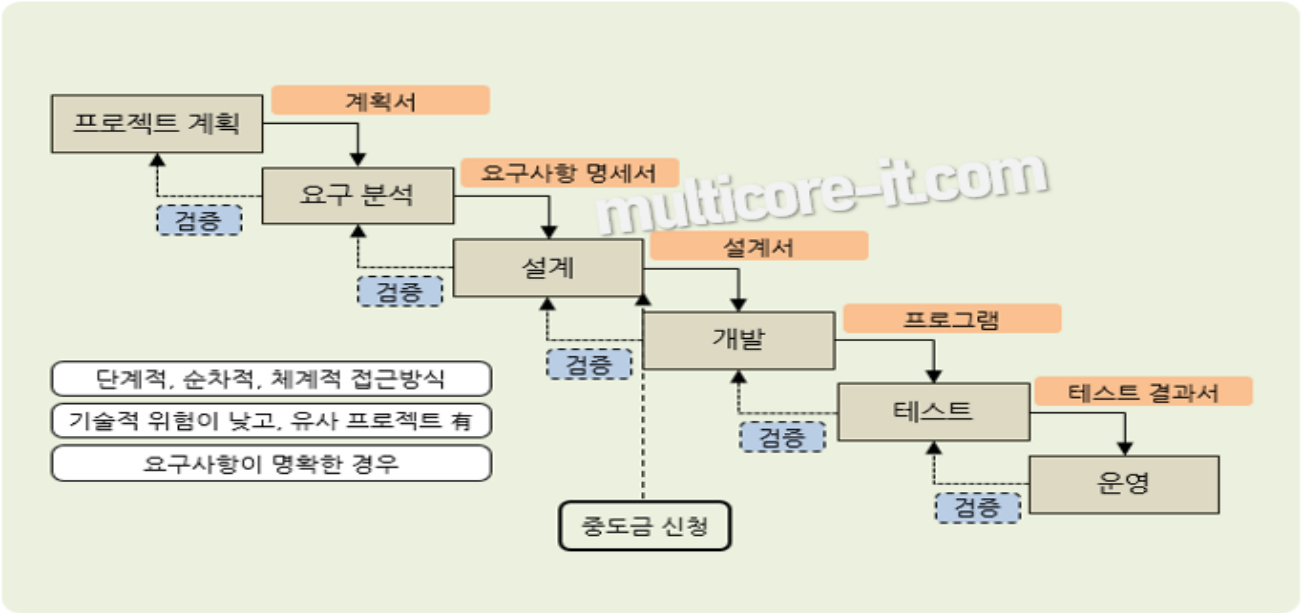
❖ 소프트웨어 생명주기

- 개발 → 유지보수 (예, windows update) → 폐기
- 최근은 지속적인 개발/배포 형태로 운영

소프트웨어 개발 모델

❖ 폭포수모델(waterfall)

- 전통적인 모델
- 순차적 개발
- 완료된 단계 이후 이전 단계의 변동이 필요한 경우 처리가 어려움



출처: https://multicore-it.com/45

- 군사용등 고비용, 고위험 소프트웨어개발에 적합

- 최근 상용소프트웨어는 빨리 배포하고 지속적으로 개선하는 형태가 바람직

폭포수 모델

1. 계획
   - 개발 범위를 결정한다.
   - 임무를 나눈다.
   - 개발계획서를 작성한다.
2. 요구분석
   - 기능적 요구사항과 비기능적 요구사항을 분석한다.
   - 요구 사항 명세서를 작성한다.
3. 설계
   - 기본설계: 소프트웨어의 전체 구조를 작성한다.
   - 상세설계: 독립적인 기능별로 나눈다.
4. 구현
   - 프로그램을 작성(코딩)한다.
5. 테스트
   - 테스트 케이스를 작성한다.
   - 개별 테스트를 진행한다.
   - 전체 테스트를 진행한다.
6. 문서화
   - 작동 사양 설명서를 작성한다.
   - 소스코드 및 테스트 결과 보고서를 작성한다.
7. 유지보수
   - 개선과 갱신을 수행한다.

단계적 모델

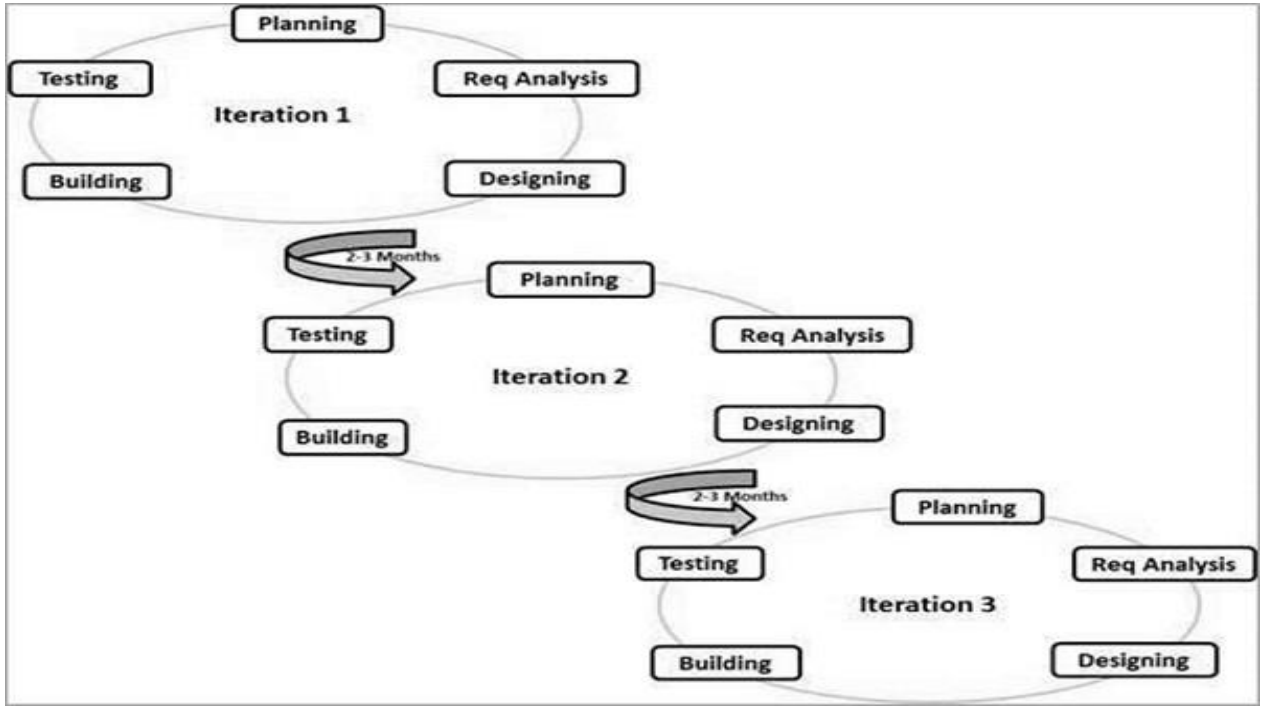
❖ 단계적(Phased) 모델

- 소프트웨어를 제공하고 사용과 개발을 동시에 진행
- 점증적(incremental) 모델
  - ✓ 단위기능별로 소프트웨어를 제공
    - 1단계: 구매 프로그램 개발
    - 2단계: 회계 프로그램 개발; 구매 프로그램 사용(유지보수)
    - 3단계: 인사관리 프로그램 개발; 회계프로그램 사용(유지보수); 구매 프로그램 사용(유지보수)
- 반복적(iterative) 모델
  - ✓ 전체기능을 간략하게 제공 (prototyping; version 1)
  - ✓ 기능과 성능을 개선한 version 2 제공
  - ✓ 개발과 배포를 반복적으로 수행하여 소프트웨어 품질 향상

애자일 모델

❖ 계속적으로 요구사항이 추가 및 변경될 수 있다는 가정하에 소프트웨어 제작

- 짧은 주기(1주 부터 2-3달까지)로 소프트웨어를 개발하고 배포(운영)
- 추가되거나 변경된 요구사항을 반영하여 개선된 소프트웨어를 개발 및 배포 (개선과 배포를 반복적으로 수행)



(출처 https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm)

설계 및 구현

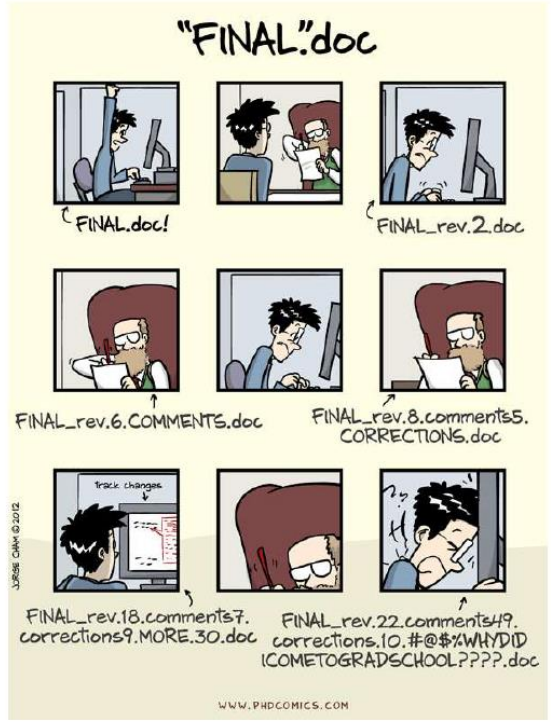
- ❖ 기본설계, 상세 설계 후에 코딩을 수행
  - 기본설계: 계층별, 모듈별로 구성 (계층, 모듈, 함수)
  - 상세설계: 함수 단위의 설계
    - ✓ 비용, 개발인력의 능력, 일정등을 고려하여 상세설계의 수준을 결정
  - 설계시는 순서도나 의사코드(pseudo code) 작성
- ❖ 설계가 상세할 수록 구현 시간과 시험 시간이 단축 (현실은 ?)

컴퓨팅사고의 중요한 개념: *추상화, 반복*

Git 과 Github

버전 관리

- 버전이란?



소프트웨어 버전 관리

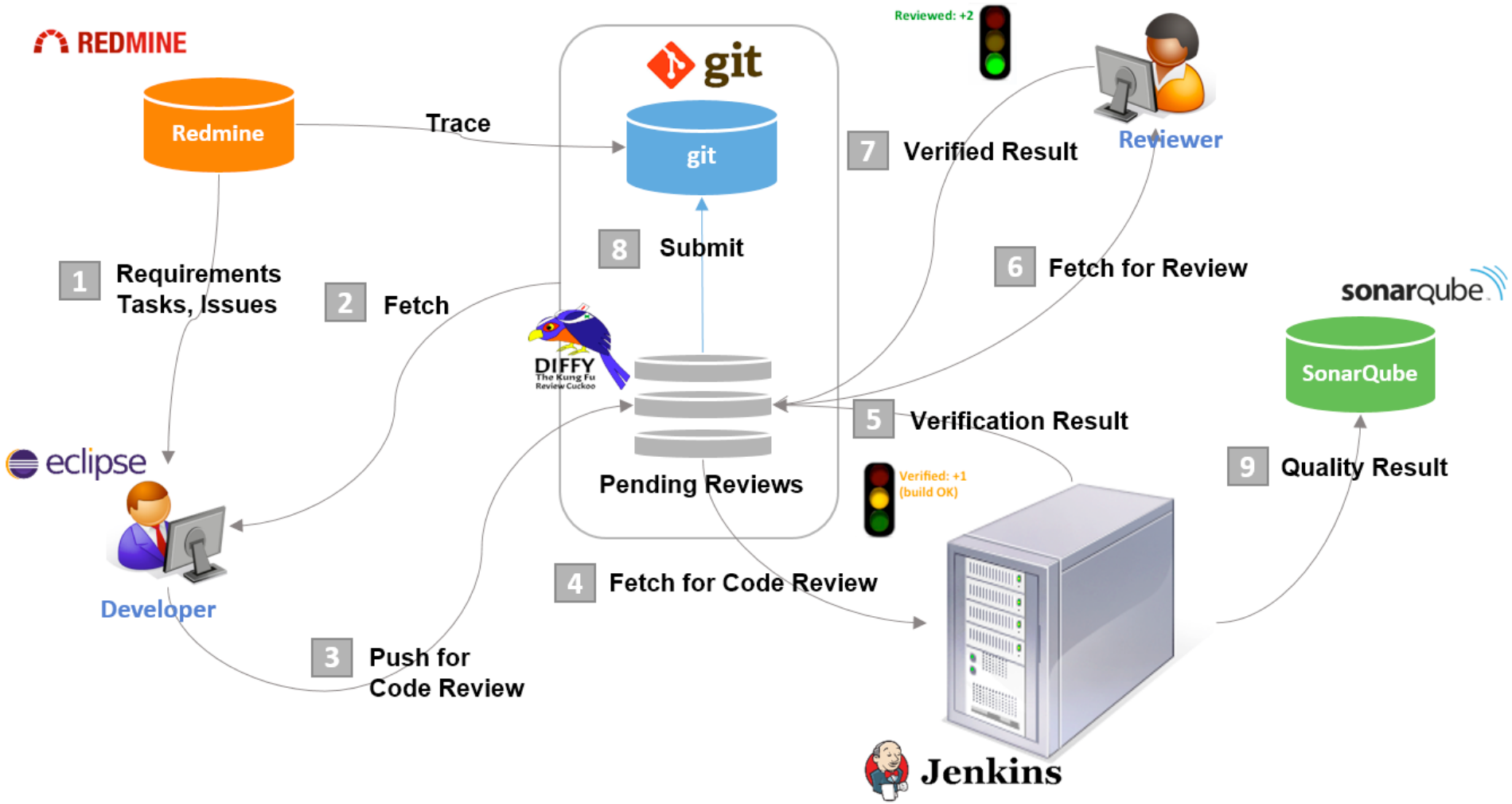
http://opentutorials.org/course/1492
http://opentutorials.org/course/1492/8035
http://software-carpentry.org/v5/novice/git/index.html

버전관리

앞의 comics와 같은 버전 관리의 한 방법…

PowerForecaster-sensys2015_v16_sec7-yk.doc	2015-04-09 오후 10:32	Microsoft Word 9...	4,156KB
PowerForecaster-sensys2015_v18.doc	2015-04-09 오후 10:11	Microsoft Word 9...	3,110KB
PowerForecaster-sensys2015_v16-sw.doc	2015-04-09 오후 9:37	Microsoft Word 9...	3,136KB
PowerForecaster-sensys2015_v16 (1).doc	2015-04-09 오후 4:23	Microsoft Word 9...	3,143KB
PowerForecaster-sensys2015_v15-sec3-sec4-sec6-...	2015-04-08 오후 6:18	Microsoft Word 9...	3,143KB
PowerForecaster-sensys2015_v15.doc	2015-04-08 오후 5:42	Microsoft Word 9...	3,146KB
PowerForecaster-sensys2015_v14.doc	2015-04-08 오후 3:53	Microsoft Word 9...	3,133KB
PowerForecaster-sensys2015_v12.doc	2015-04-08 오후 3:11	Microsoft Word 9...	3,152KB
PowerForecaster-sensys2015_v13.doc	2015-04-08 오후 3:11	Microsoft Word 9...	3,152KB
PowerForecaster-sensys2015_v11_sec5_yk.doc	2015-04-08 오전 2:45	Microsoft Word 9...	3,134KB
PowerForecaster-sensys2015_v11.doc	2015-04-08 오전 12:29	Microsoft Word 9...	3,136KB
PowerForecaster-sensys2015_v10-swkang.doc	2015-04-08 오전 12:25	Microsoft Word 9...	3,143KB
PowerForecaster-sensys2015_v9 (1).doc	2015-04-08 오전 12:11	Microsoft Word 9...	3,146KB
PowerForecaster-sensys2015_v10.doc	2015-04-07 오후 11:08	Microsoft Word 9...	3,148KB
PowerForecaster-sensys2015_v8_is.doc	2015-04-07 오전 3:59	Microsoft Word 9...	4,361KB
PowerForecaster-sensys2015_papercheck_sec4_yk....	2015-04-07 오전 12:21	Microsoft Word 9...	3,136KB

Typical ALM with Open Source



소프트웨어 버전 관리

❖ 자신의 작업 진행사항을 추적하고, 다른 사람과 협업할 수 있는 시스템

- 소프트웨어 뿐만 아니라 시간에 따라 변화가 생기고, 다른 사람과 공유라는 모든 일에 버전 관리 적용가능
  - ✓ 책, 논문, 작은 데이터 셋 등

❖ **version control**: A tool for managing changes to a set of files. Each set of changes creates a new revision of the files; the version control system allows users to recover old revisions reliably, and helps manage conflicting changes made by different users

- Version 관리 시스템에 제출(committ)된 것은 잃어버리지 않는다. 즉 이전에 제출된 것들을 찾아보고 복원할 수도 있다.
- 누가, 언제, 무엇을 바꾸었는지 기록들을 유지한다.
- 협업하는 사람들에게 충돌이 발생한 지점을 알려준다.

버전관리시스템 (git, github)

버전 관리

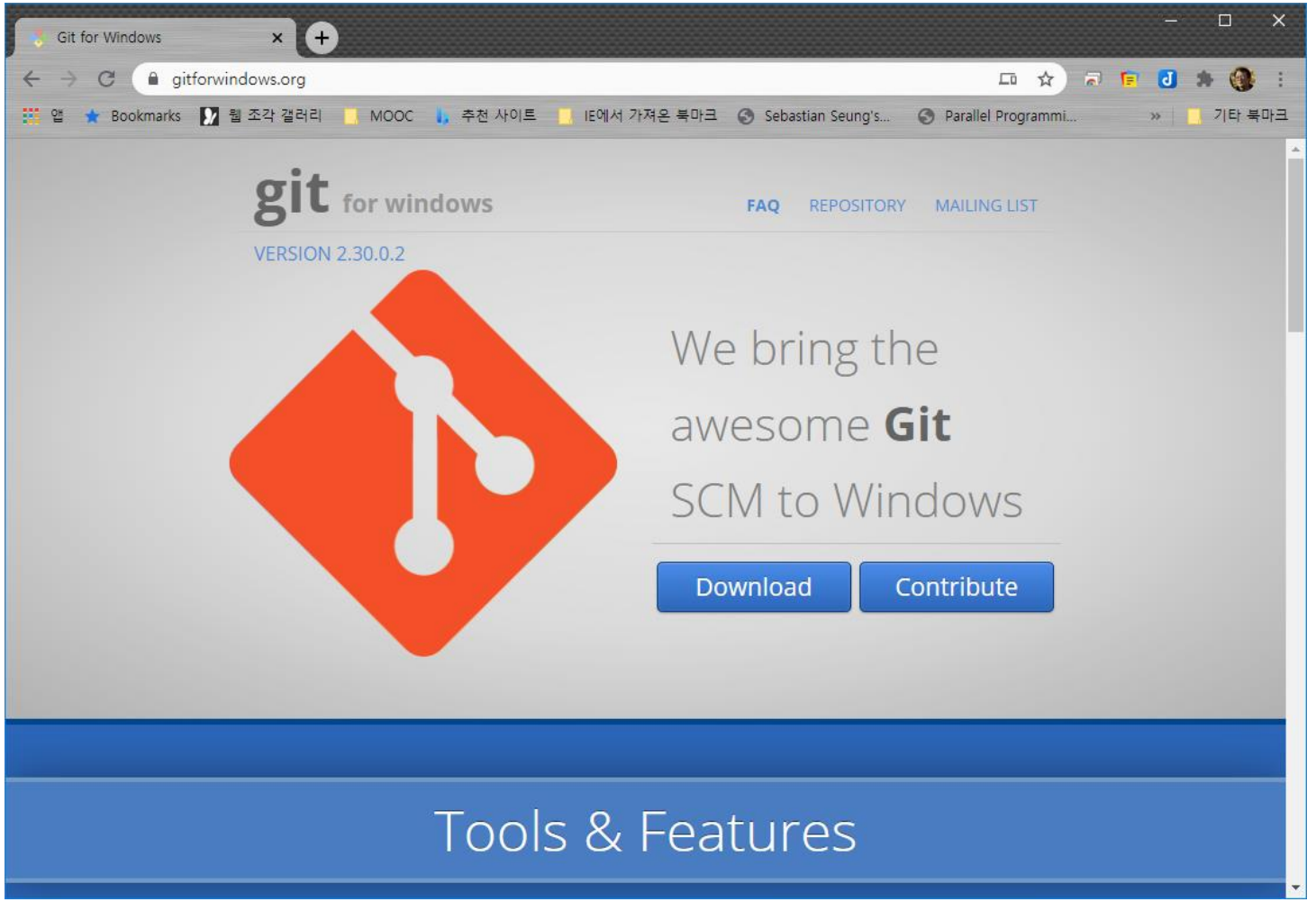
- 수십 명, 수백 명이 만들고 업데이트 하는 소프트웨어를 어떻게 관리할까?

→ 버전 관리 시스템이 필요

- 버전 관리 시스템
  - 그 중 하나인 **git**에 대해 알아보고 직접 **git**을 사용해본다
    - 리누스 토발즈가 linux kernel 소스 관리를 위해 만든 소스 버전 관리 프로그램
    - Git은 속도에 중점을 둔 분산형 버전관리 시스템(DVCS)
    - 대형 프로젝트에서 효과적이고 유용
  - github
    - github.com : git을 사용하는 프로젝트 용 웹 호스팅 서비스
    - git 원격 저장소를 만들고 다른 사람들과 공유할 수 있음
    - 오픈소스는 무료 서비스

Git 설치 및 이용

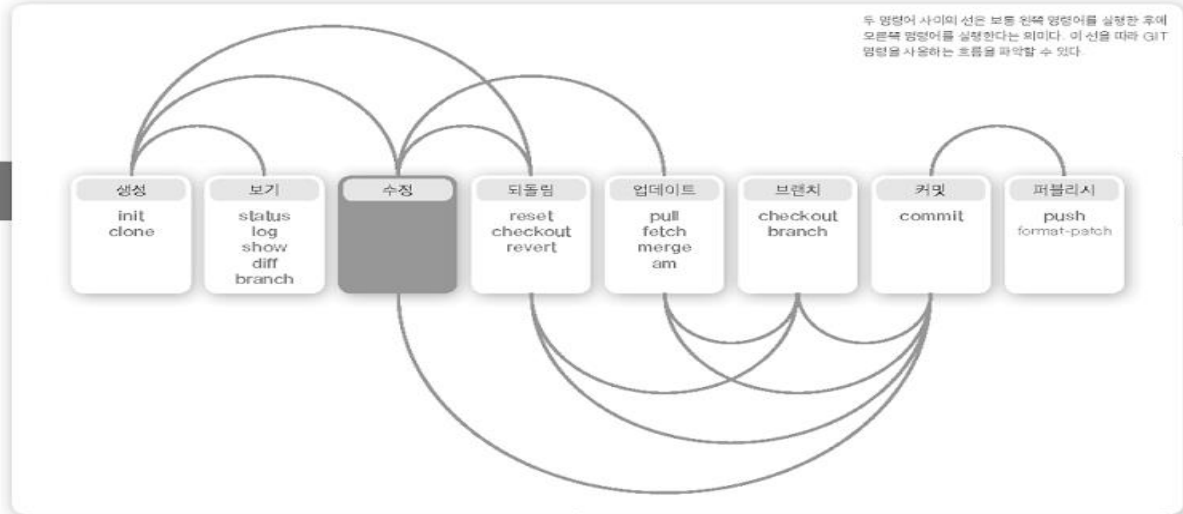
❖ 윈도우 환경 (https://gitforwindows.org/)



KOREATECH 한국기술교육대학교

GIT 작업 흐름과 명령어

도움말: git "명령어" --help
Git의 글로벌 설정은 $HOME/.gitconfig에 저장 (git config --help)



생성

새 저장소 생성하기
```
cd ~/projects/myproject
git init
git add .
```

기존 저장소 Clone하기
```
git clone ~/existing/repo ~/new/repo
git clone git://host.org/project.git
git clone you@host.org/project.git
```

보기

워킹 디렉터리의 파일 상태 보기
```
git status
```

파일의 변경사항 보기
```
git diff
```

$ID1과 $ID2 사이의 변경사항 보기
```
git diff $id1 $id2
```

커밋 히스토리 보기
```
git log
```

특정 파일의 커밋 히스토리별 변경사항 보기
```
git log -p $file $dir/ec/tory/
```

특정 파일을 누가 언제 고쳤는지 보기
```
git blame $file
```

$ID 커밋 보기
```
git show id
```

$ID 버전의 파일 보기
```
git show $id:$file
```

로컬 브랜치들 보기
```
git branch
```
('*' 표시는 현재 브랜치를 나타냄)

범례

$id - 커밋 ID, 브랜치 이름, 태그 이름을 나타냄
$file - 파일 이름
$branch - 브랜치 이름

개념

Git 기본

master : 기본 브랜치
origin : 기본 리모트 저장소
HEAD : 현재 브랜치
HEAD^ : HEAD의 부모
HEAD~4 : HEAD의 부모의 부모의 부모의 부모

되돌림

마지막 커밋 시점으로 되돌리기
```
git reset --hard
```
▲Hard Reset은 되돌릴 수 없음

마지막 커밋 내용을 되돌리고 커밋하기
```
git revert HEAD
```
새로운 커밋 생성

특정 커밋 내용을 되돌리고 커밋하기
```
git revert $id
```
새로운 커밋 생성

마지막 커밋 수정하기
```
git commit -a --amend
```
(잘못 커밋해서 수정하고 싶을 때)

$id 시점의 파일을 꺼내기
```
git checkout $id $file
```

브랜치

브랜치를 Checkout하기
```
git checkout $id
```

$branch1을 $branch2에 Merge하기
```
git checkout $branch2
git merge $branch1
```

새 브랜치 만들기
```
git branch $branch
```

$other와 같은 커밋을 가리키는 브랜치를 새로 만들고 바로 Checkout하기
```
git checkout -b $new_branch $other
```

$branch를 삭제하기
```
git branch -d $branch
```

업데이트

origin에서 최신 데이터를 가져오기
```
git fetch
```
(Merge하지는 않음)

origin에서 최신 데이터를 가져와 Merge하기
```
git pull
```
(Fetch하고 Merge까지 함)

누군가 보낸 패치를 Merge하기
```
git am -3 patch.mbox
```
(충돌이 발생하면 해결 후 git am --resolved)

퍼블리시

현재 모든 수정사항을 커밋하기
```
git commit -a
```

다른 개발자에게 보낼 패치를 작성하기
```
git format-patch origin
```

origin으로 업데이트를 Push하기
```
git push
```

버전이나 마일스톤을 생성하기
```
git tag v1.0
```

유용한 명령어

문제가 발생한 커밋 이진탐색하기
```
git bisect start (이진탐색 시작)
git bisect good $id ($id를 문제없는 상태로 표시)
git bisect bad $id ($id를 문제 있는 상태로 표시)

git bisect bad/good (현재 상태가 문제가 있는지 없는지 설정)
git bisect visualize (gitk를 실행하여 확인함)
git bisect reset (시작했던 상태로 Checkout 함)
```

저장소의 무결성 검사 및 저장소 청소하기
```
git fsck
git gc --prune
```

워킹 디렉터리에서 'foo()'라는 문자열 검색하기
```
git grep "foo()"
```

Merge 충돌 해결

Merge 충돌 내용 보기
```
git diff (충돌 내용 전체 보기)
git diff --base $file (Merge Base를 기준으로)
git diff --ours $file (현 브랜치를 기준으로)
git diff --theirs $file (Merge할 브랜치를 기준으로)
```

충돌이 생기는 패치 버리기
```
git reset --hard
git rebase --skip
```

충돌 해결 후 Merge 진행하기
```
git add $conflicting_file (충돌 해결한 파일)
git rebase --continue
```

GIT정리

- “git config” configures a user name, email address, editor, and other preferences once per machine.
- “git init” initializes a repository.
- “git status” shows the status of a repository.

Files can be stored in a project's working directory (which users see),
the staging area (where the next commit is being built up) and
the local repository (where snapshots are permanently recorded).

- “git add” puts files in the staging area.
- “git commit” creates a snapshot of the staging area in the local repository.
- “git diff” displays differences between revisions.
- “git checkout” recovers old versions of files.
- The .gitignore file tells Git what files to ignore.

저장소를 만드는 다른 방법

❖ 이미 만들어진 저장소를 복제하여 사용할 수 있다.

```
git clone <repo>
```

- 저장소 <repo>를 사용자 컴퓨터에 복제
- <repo>는 로컬 파일시스템에 존재하거나 원격지에 있어서 HTTP나 SSH로 접근
- 로컬 컴퓨터의 특정 디렉토리로 복제할 경우 <directory> 명시

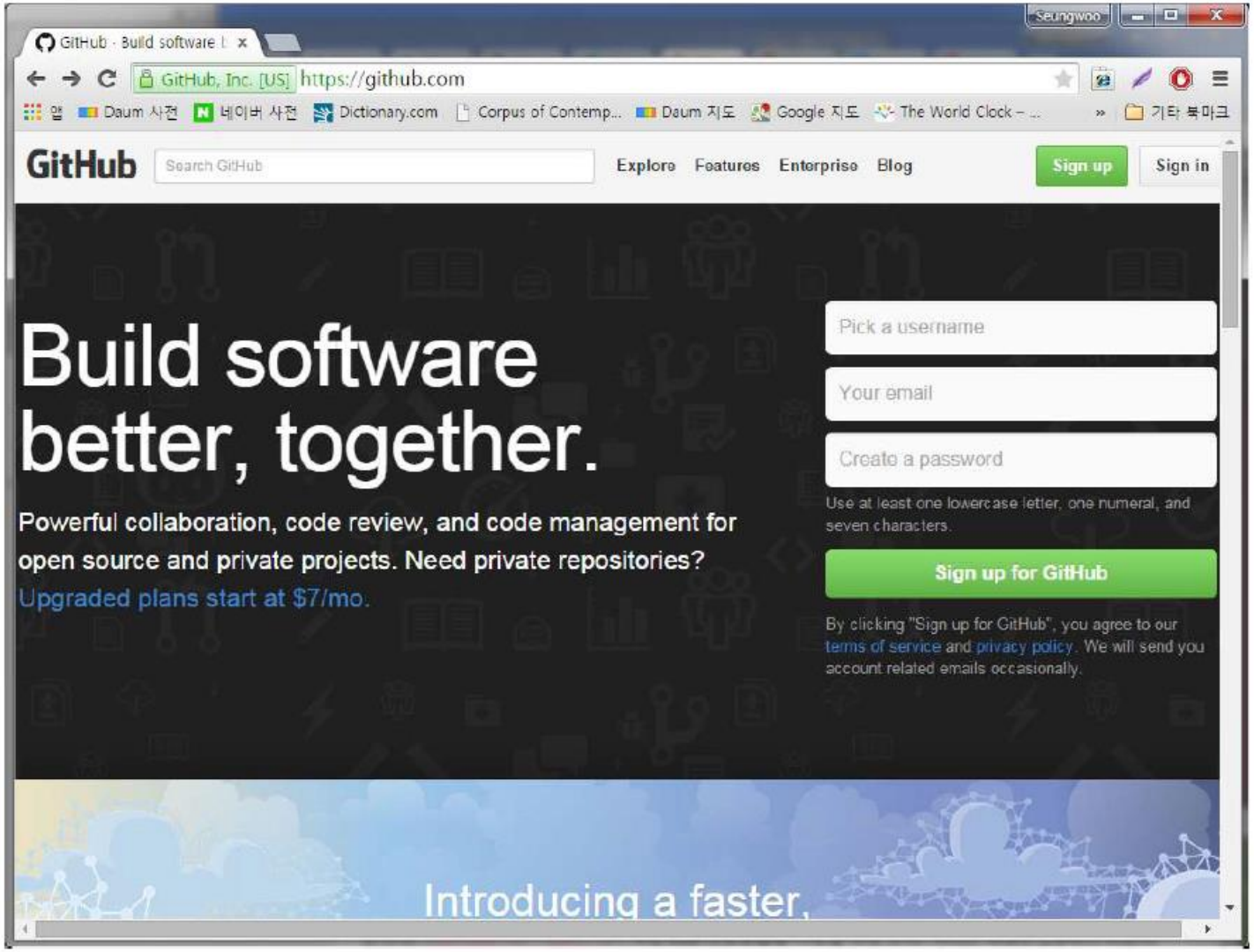
```
git clone <repo> <directory>
```

❖ 예제

```
$ git clone ssh://john@example.com/path/to/my-project.git
$ cd my-project
$ # Start working on the project

$ git clone https://github.com/path/to/my-project
$ cd my-project
$ # Start working on the project
```

Github 연동 (협업)



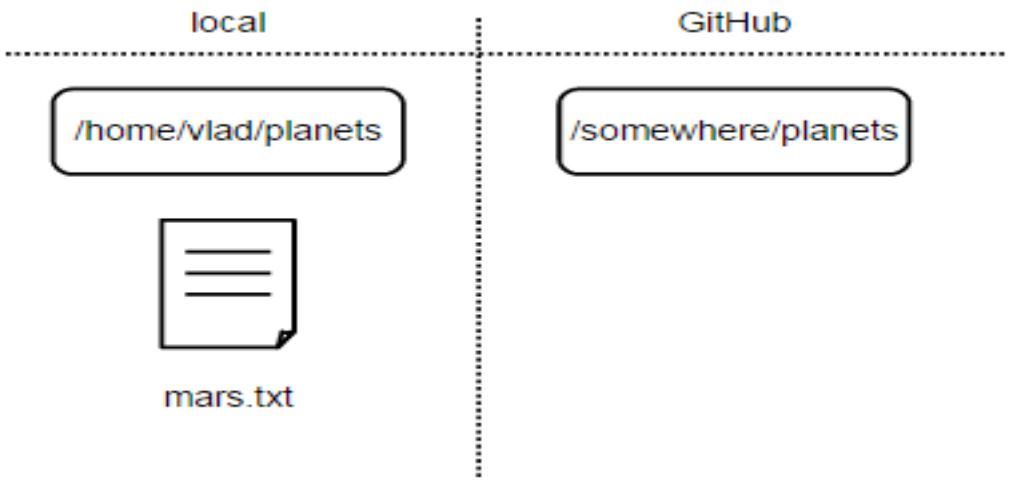
KOREATECH 한국기술교육대학교

협업(Collaborating)

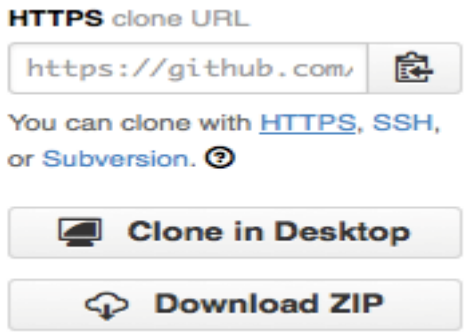
- 로컬머쉰

```
$ mkdir planets
$ cd planets
$ git init
```

- 로컬 파일 mars.txt 생성



- 두 저장소를 연결
- 복제에 사용할 프로토콜을 ssh대신 https로 변경



KOREATECH 한국기술교육대학교

협업(Collaborating)

❖ “git remote” 명령으로 외부 저장소와 연동

```
$ git remote add origin https://github.com/vlad/planets
```

- 해당 url을 자신의 계정으로 변경해서 사용

❖ 원격 연결할 수 있는 외부 저장소와 url을 보려면

```
$ git remote –v
origin https://github.com/vlad/planets.git (push)
origin https://github.com/vlad/planets.git (fetch)
```

✓ 외부 저장소에 대한 로컬 nickname으로 origin사용

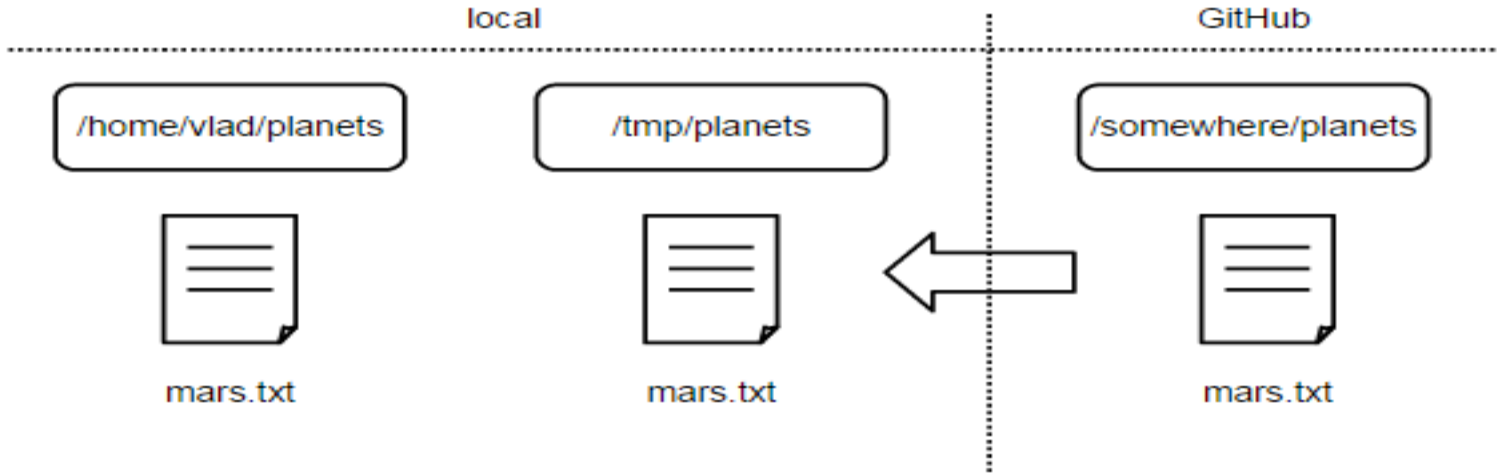
❖ 로컬 저장소의 변경사항들을 원격 저장소에 전송 (push)

```
$ git push origin master
Counting objects: 9, done.
Delta compression using up to 4 threads.
……
```

협업(Collaborating)

❖ 원격 저장소를 활용를 활용해서 새로운 로컬 저장소 생성 (git clone)

```
$ cd /tmp
$ git clone https://github.com/vlad/planets.git
```



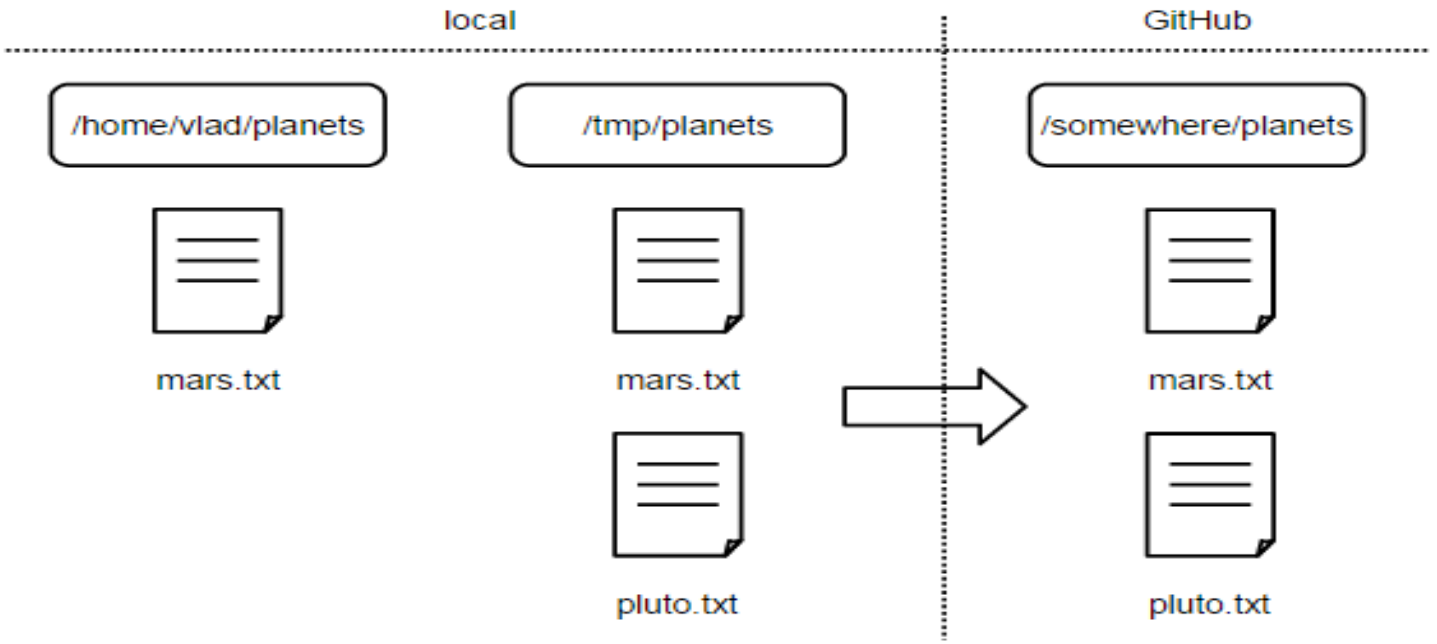
❖ /tmp/planets 저장소에 pluto.txt 파일을 추가

```
$ cd /tmp/planets
$ nano pluto.txt
$ cat pluto.txt
It is so a planet!
```

협업(Collaborating)

❖ 파일 추가된 사항을 반영

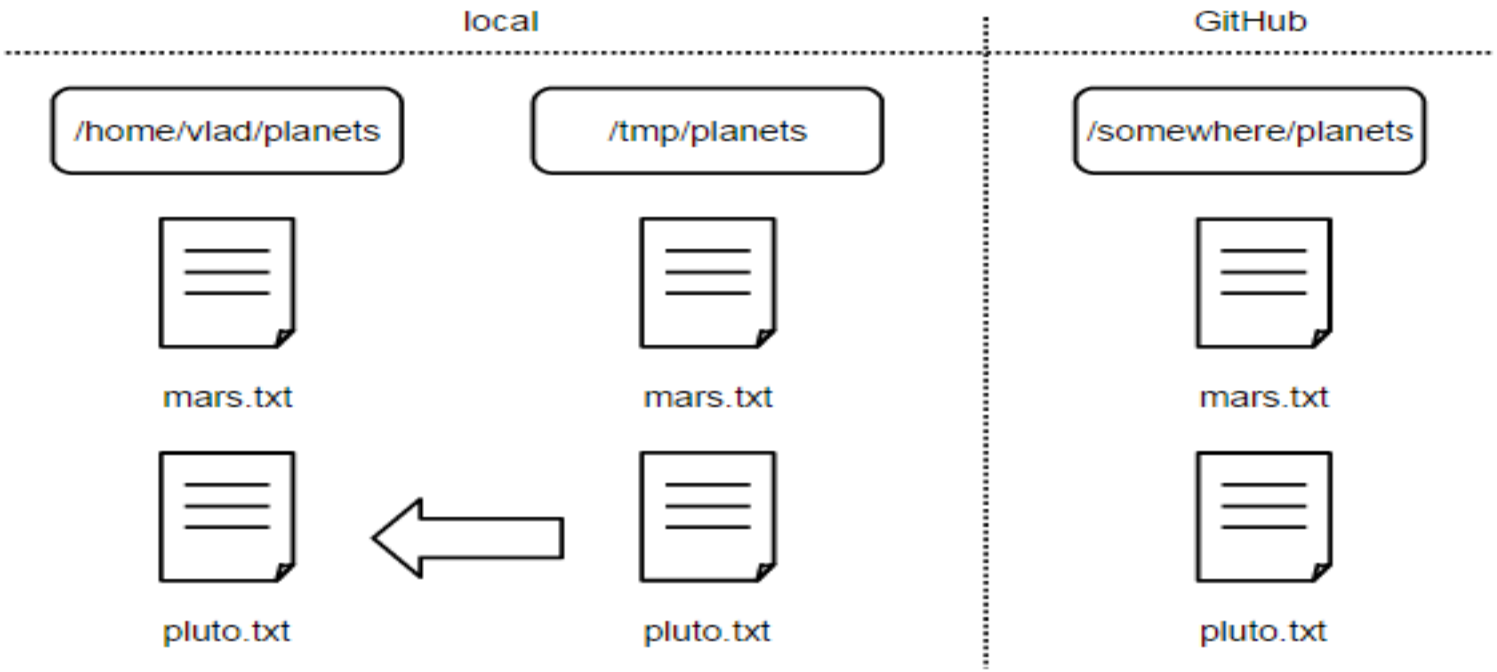
```
$ git add pluto.txt
$ git commit -m "Some notes about Pluto"
1 file changed, 1 insertion(+)
create mode 100644 pluto.txt
$ git push origin master
……
```



협업(Collaborating)

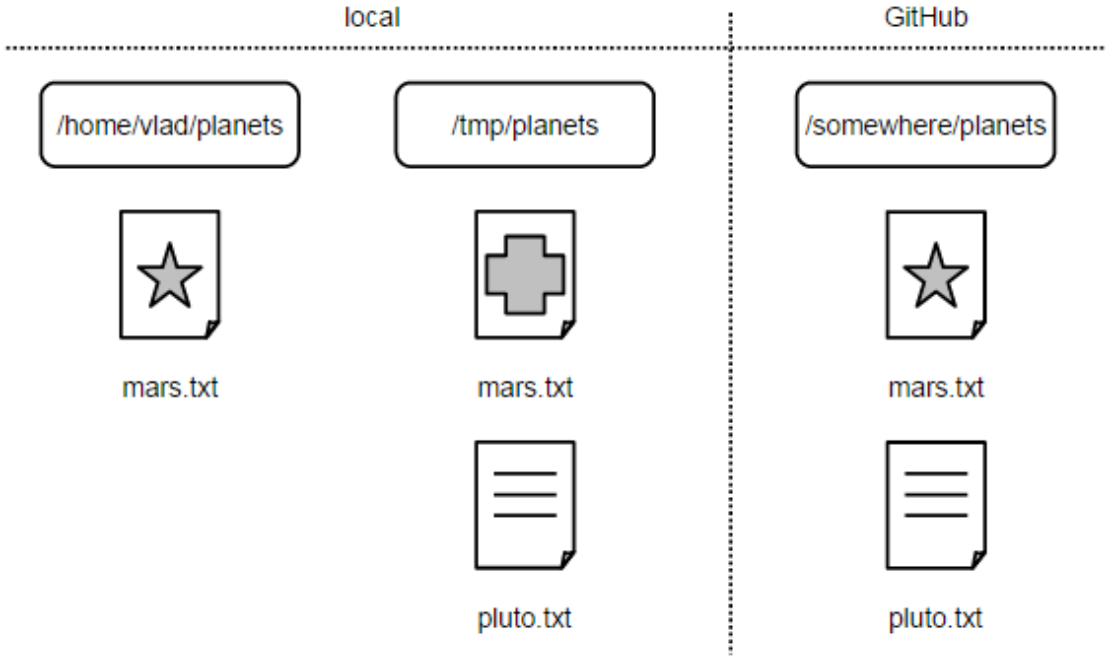
❖ 원격 저장소의 변경사항을 원래 저장소에 반영

```
$ cd ~/planets
$ git pull origin master
……
```



충돌(Conflicts)

- 여러 사람이 동시에 작업을 할 때 같은 파일을 동시에 수정하는 경우가 발생할 수 있다.
- ~/planets 와 /tmp/planets 의 mars.txt 파일을 서로 다르게 수정
- /tmp/planets에서 push 시도하면 충돌 오류 발생
- 충돌을 해결하기 위해서 원격저장소의 파일을 가져와서 conflict가 발생한 파일을 수정하고 원격 저장소에 다시 파일을 올린다



Git 정리

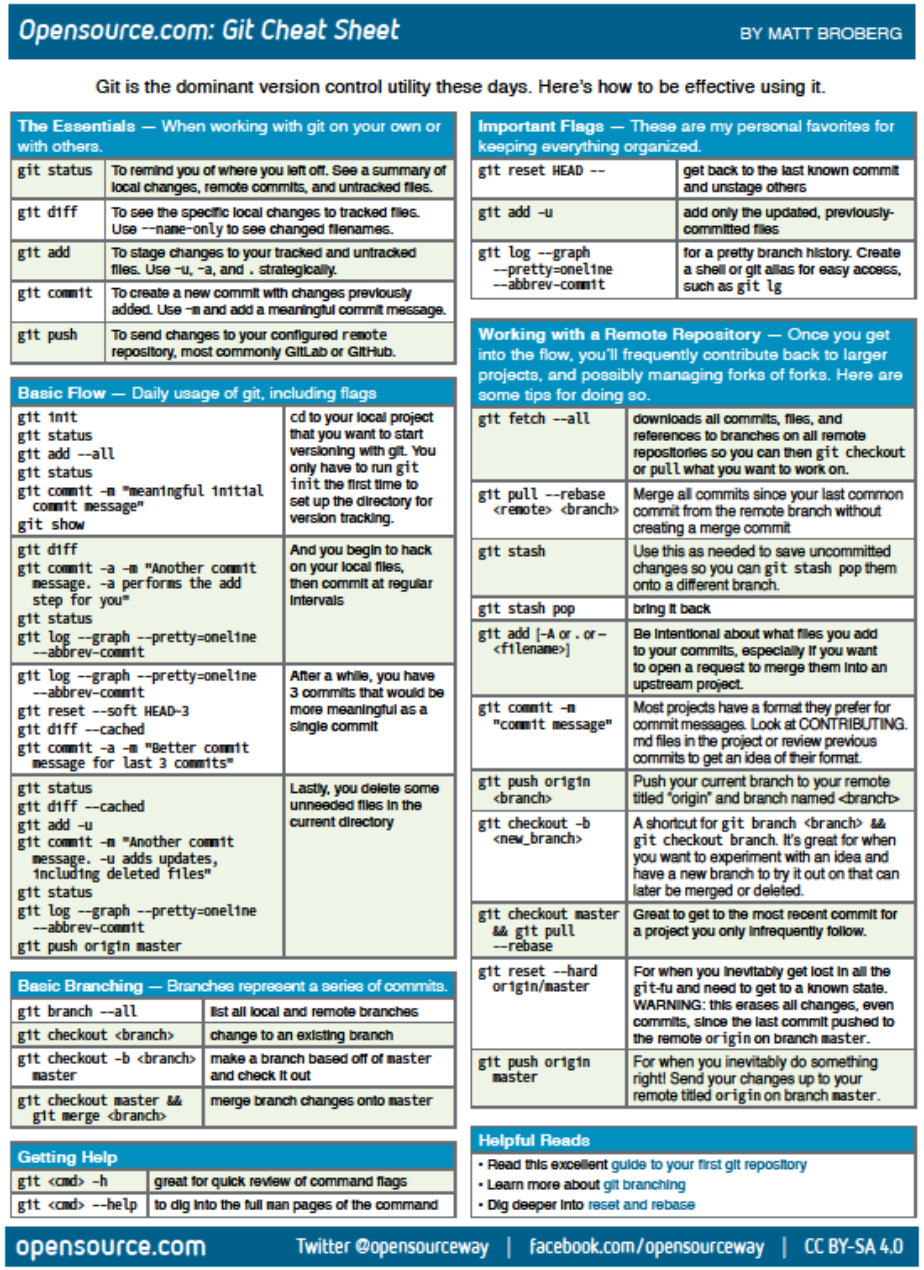
- 버전관리를 위해 git과 github 사용법 정리 (http://swcarpentry.github.io/git-novice/)
- 오래전부터 버전관리를 위해 rcs, cvs, svn 등이 존재
- 오픈소스 호스팅을 위해 github 외에도 sourceforge.net, savannah.gnu.org, code.google.com 등이 사용되고 있다.
- Github이 일반적으로 사용됨

참고자료

https://opensource.com/resources/what-is-git

https://opensource.com/article/19/5/practical-learning-exercise-git

https://guides.github.com/activities/hello-world/



Opensource.com: Git Cheat Sheet BY MATT BRODERG

Git is the dominant version control utility these days. Here's how to be effective using it.

The Essentials — When working with git on your own or with others.

`git status`	To remind you of where you left off. See a summary of local changes, remote commits, and untracked files.
`git diff`	To see the specific local changes to tracked files. Use `--name-only` to see changed filenames.
`git add`	To stage changes to your tracked and untracked files. Use `-u`, `-a`, and `.` strategically.
`git commit`	To create a new commit with changes previously added. Use `-m` and add a meaningful commit message.
`git push`	To send changes to your configured remote repository, most commonly GitLab or GitHub.

Basic Flow — Daily usage of git, including flags

`git init`
`git status`
`git add --all`
`git status`
`git commit -m "meaningful initial commit message"`
`git show`
	cd to your local project that you want to start versioning with git. You only have to run `git init` the first time to set up the directory for version tracking.

`git diff`
`git commit -a -m "Another commit message. -a performs the add step for you"`
`git status`
`git log --graph --pretty=oneline --abbrev-commit`
	And you begin to hack on your local files, then commit at regular intervals

`git log --graph --pretty=oneline --abbrev-commit`
`git reset --soft HEAD~3`
`git diff --cached`
`git commit -a -m "Better commit message for last 3 commits"`
	After a while, you have 3 commits that would be more meaningful as a single commit

`git status`
`git diff --cached`
`git add -u`
`git commit -m "Another commit message. -u adds updates, including deleted files"`
`git status`
`git log --graph --pretty=oneline --abbrev-commit`
`git push origin master`
	Lastly, you delete some unneeded files in the current directory

Basic Branching — Branches represent a series of commits.

`git branch --all`	list all local and remote branches
`git checkout <branch>`	change to an existing branch
`git checkout -b <branch> master`	make a branch based off of master and check it out
`git checkout master && git merge <branch>`	merge branch changes onto master

Getting Help

`git <cmd> -h`	great for quick review of command flags
`git <cmd> --help`	to dig into the full man pages of the command

Important Flags — These are my personal favorites for keeping everything organized.

`git reset HEAD --`	get back to the last known commit and unstage others
`git add -u`	add only the updated, previously-committed files
`git log --graph --pretty=oneline --abbrev-commit`	for a pretty branch history. Create a shell or git alias for easy access, such as `git lg`

Working with a Remote Repository — Once you get into the flow, you'll frequently contribute back to larger projects, and possibly managing forks of forks. Here are some tips for doing so.

`git fetch --all`	downloads all commits, files, and references to branches on all remote repositories so you can then `git checkout` or `pull` what you want to work on.
`git pull --rebase <remote> <branch>`	Merge all commits since your last common commit from the remote branch without creating a merge commit
`git stash`	Use this as needed to save uncommitted changes so you can `git stash pop` them onto a different branch.
`git stash pop`	bring it back
`git add [-A or . or -- <filename>]`	Be intentional about what files you add to your commits, especially if you want to open a request to merge them into an upstream project.
`git commit -m "commit message"`	Most projects have a format they prefer for commit messages. Look at CONTRIBUTING.md files in the project or review previous commits to get an idea of their format.
`git push origin <branch>`	Push your current branch to your remote titled "origin" and branch named <branch>
`git checkout -b <new_branch>`	A shortcut for `git branch <branch> && git checkout branch`. It's great for when you want to experiment with an idea and have a new branch to try it out on that can later be merged or deleted.
`git checkout master && git pull --rebase`	Great to get to the most recent commit for a project you only infrequently follow.
`git reset --hard origin/master`	For when you inevitably get lost in all the git-fu and need to get to a known state. WARNING: this erases all changes, even commits, since the last commit pushed to the remote `origin` on branch `master`.
`git push origin master`	For when you inevitably do something right! Send your changes up to your remote titled `origin` on branch `master`.

Helpful Reads

- Read this excellent guide to your first git repository
- Learn more about git branching
- Dig deeper into reset and rebase

opensource.com Twitter @opensourceway | facebook.com/opensourceway | CC BY-SA 4.0

KOREATECH 한국기술교육대학교