**Configuration Management Developer Guide for Git**

**스마트워크 2,3단계 구축\_형상관리\_가이드**

****Version 1.0

2016.09.05

작성자 : 문해민



**Table of Contents**

[**1.** **개요** - 6 -](#_Toc461182176)

[**2.** **형상관리 작업 환경 구축 및 절차** - 6 -](#_Toc461182177)

[**2.1.** **시스템 요구사항** - 6 -](#_Toc461182178)

[**2.2.** **버전 관리 환경 구축** - 7 -](#_Toc461182179)

[**2.2.1.** **Eclipse Plug-In – 필수** - 7 -](#_Toc461182180)

[**2.2.2.** **Repository Clone** - 10 -](#_Toc461182181)

[**2.2.3.** **Git Bash [ CLI ] – 선택 사항** - 14 -](#_Toc461182182)

[**2.3.** **문서 기록 및 커뮤니케이션 방식 [ 이슈 관리 ]** - 15 -](#_Toc461182183)

[**3.** **Eclipse Git Plug-in 매뉴얼** - 16 -](#_Toc461182184)

[**3.1.** **User Interface** - 16 -](#_Toc461182185)

[**3.2.** **Commit** - 19 -](#_Toc461182186)

[**3.3.** **Push** - 20 -](#_Toc461182187)

[**3.4.** **Pull** - 22 -](#_Toc461182188)

[**3.5.** **Branch** - 23 -](#_Toc461182189)

[**3.5.1.** **Branch 생성** - 23 -](#_Toc461182190)

[**3.5.2.** **Branch 이동** - 24 -](#_Toc461182191)

[**3.5.3.** **Branch 병합** - 25 -](#_Toc461182192)

[**4.** **스마트 워크 2, 3단계 프로젝트 버전관리 규약** - 26 -](#_Toc461182193)

[**4.1.** **작업 시작** - 27 -](#_Toc461182194)

[**4.2.** **Commit** - 27 -](#_Toc461182195)

[**4.3.** **Branch** - 27 -](#_Toc461182196)

[**4.4.** **Push** - 28 -](#_Toc461182197)

[**4.5.** **Conflict [ 장애 상황 발생 시 ]** - 28 -](#_Toc461182198)

[**4.5.1.** **같은 부분을 수정했을 경우** - 28 -](#_Toc461182199)

[**4.5.2.** **최신 원격 저장소의 내용을 반영하지 않았을 경우** - 33 -](#_Toc461182209)

[**5.** **학습자료 링크** - 33 -](#_Toc461182210)

[**5.1.** **누구나 쉽게 이해할 수 있는 Git 입문** - 33 -](#_Toc461182211)

[**5.2.** **Pro Git** - 33 -](#_Toc461182212)

[**5.3.** **Egit( Eclipse Git plug-in ) Document** - 33 -](#_Toc461182213)

[**5.4.** **SVN 능력자를 위한 Git 개념 가이드** - 33 -](#_Toc461182214)

[**6.** **별첨** - 34 -](#_Toc461182215)

[**6.1.** **Git Flow [ Branch Work Flow ]** - 34 -](#_Toc461182216)

1. **개요**

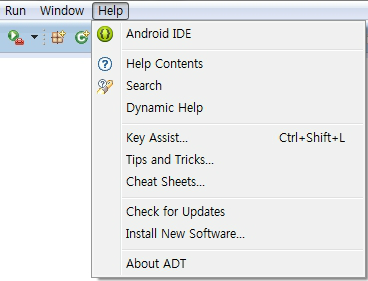
**Abstract**

본 문서는 ‘한화 S&C의 스마트워크 2, 3단계 롤 아웃 프로젝트’에 적용되는 형상관리 개발자 가이드 문서 입니다. 프로젝트의 원활한 협업을 위한 형상관리 환경 구축, 커밋, 브랜치 규약 등을 설명합니다.

1. **형상관리 작업 환경 구축 및 절차**
   1. **버전 관리 환경 구축**
      1. **Eclipse Plug-In – 필수**

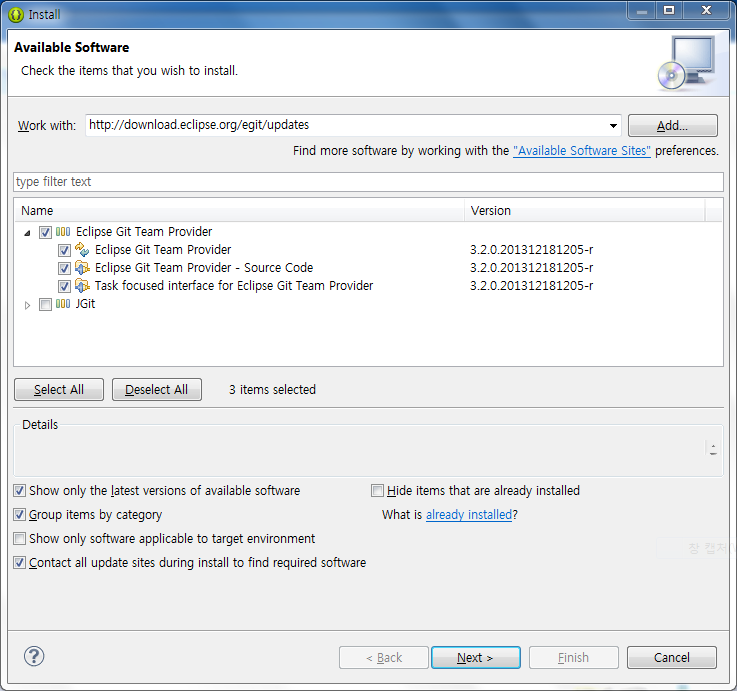
손쉬운 Git 환경을 구축하기 위해, 본 프로젝트의 Base IDE인 Eclipse에 Plug-in 방식으로 Git 환경을 구축할 것입니다.

* 이클립스를 실행 후 상위 탭에서 Help -> Install New Software를 클릭 합니다.

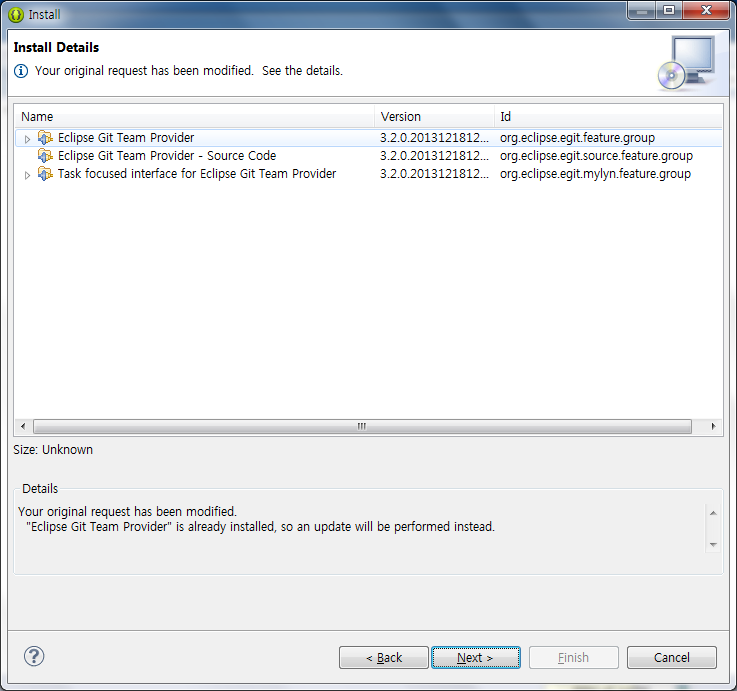


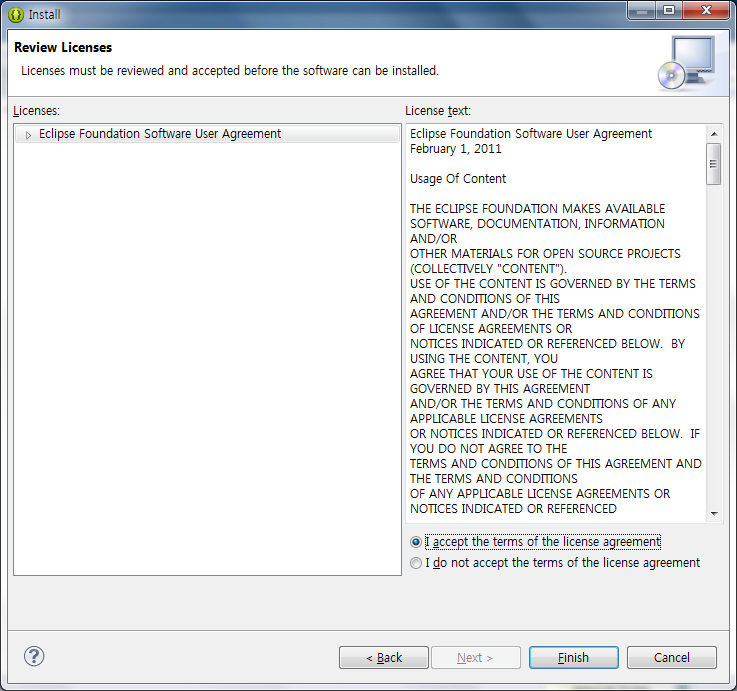
* Work with에 다음과 같은 URL을 입력 합니다.

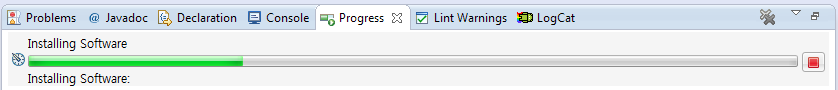
<http://download.eclipse.org/egit/updates>



* Eclipse Git Team Provider 부분을 체크한 후 Next 버튼을 눌러 설치를 진행합니다.

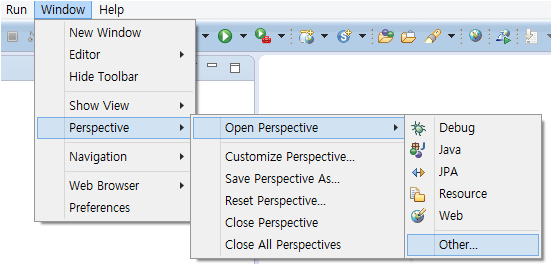




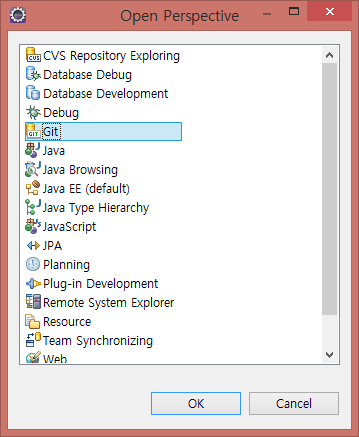


* 설치가 완료된 후 Eclipse를 재 시작 합니다.
* Eclipse가 재 기동 된 후 상단 탭의

**Windows – Perspective – Open Perspective – Other**로 이동합니다.



* 다음과 같은 목록에서 Git을 선택한 후 OK 버튼을 누릅니다.

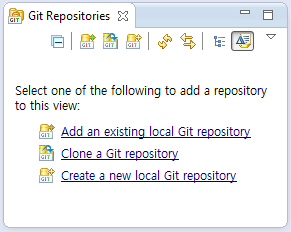
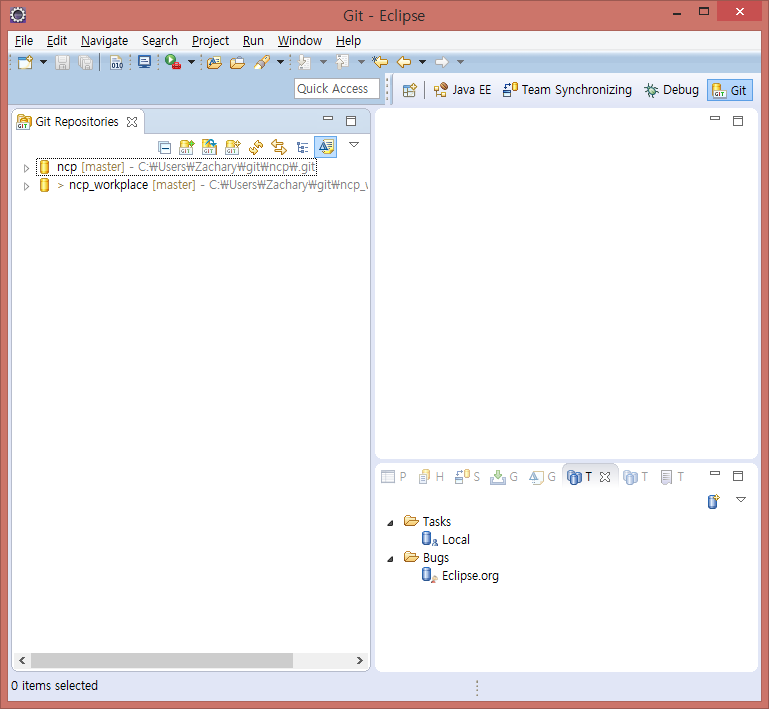


* + 1. **Repository Clone**

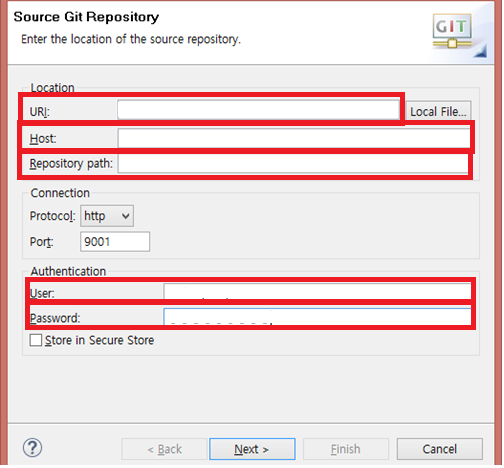
Repository를 복제하기 위해선, Git 사용자 계정이 필요합니다. 이번 Chapter를 진행하기 전 **서버 관리자에게 ‘ID/PW’ 형태로 계정을 신청**하시길 바랍니다.

**서버 관리자 : OOO, OOO**

* Git Repositories – Clone a Git repository 아이콘을 클릭합니다.



* 다음과 같은 창에 Git Server에 대한 정보를 입력 후, Next 버튼을 클릭합니다



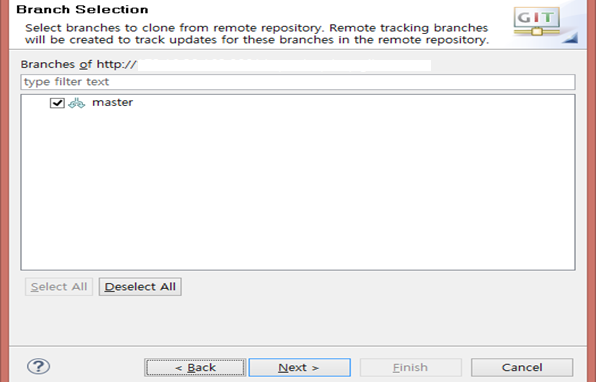
**URI : 문의**

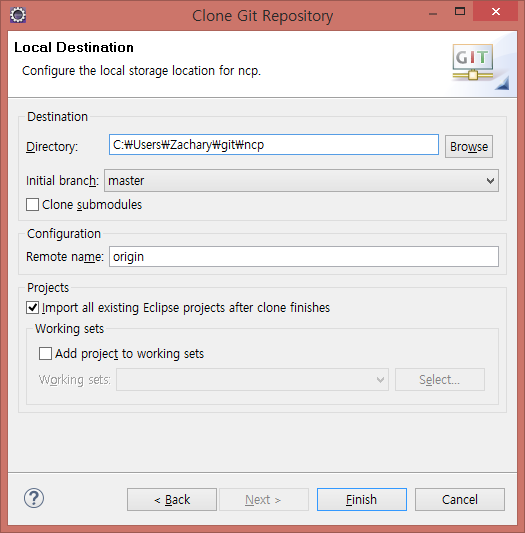
**[ ncp\_workplace의 경우 해당 경로를 입력하십시오. ]**

**Host : 문의**

**Repository path : 문의**

**User/Password : 서버 관리자로부터 승인이 완료된 자신의 계정 정보**





**Branch Setting과 Projects Setting까지 완료되면 프로젝트 목록에**

**Ncp 프로젝트 or ncp\_workplace 프로젝트가 추가되어 있을 것입니다.**

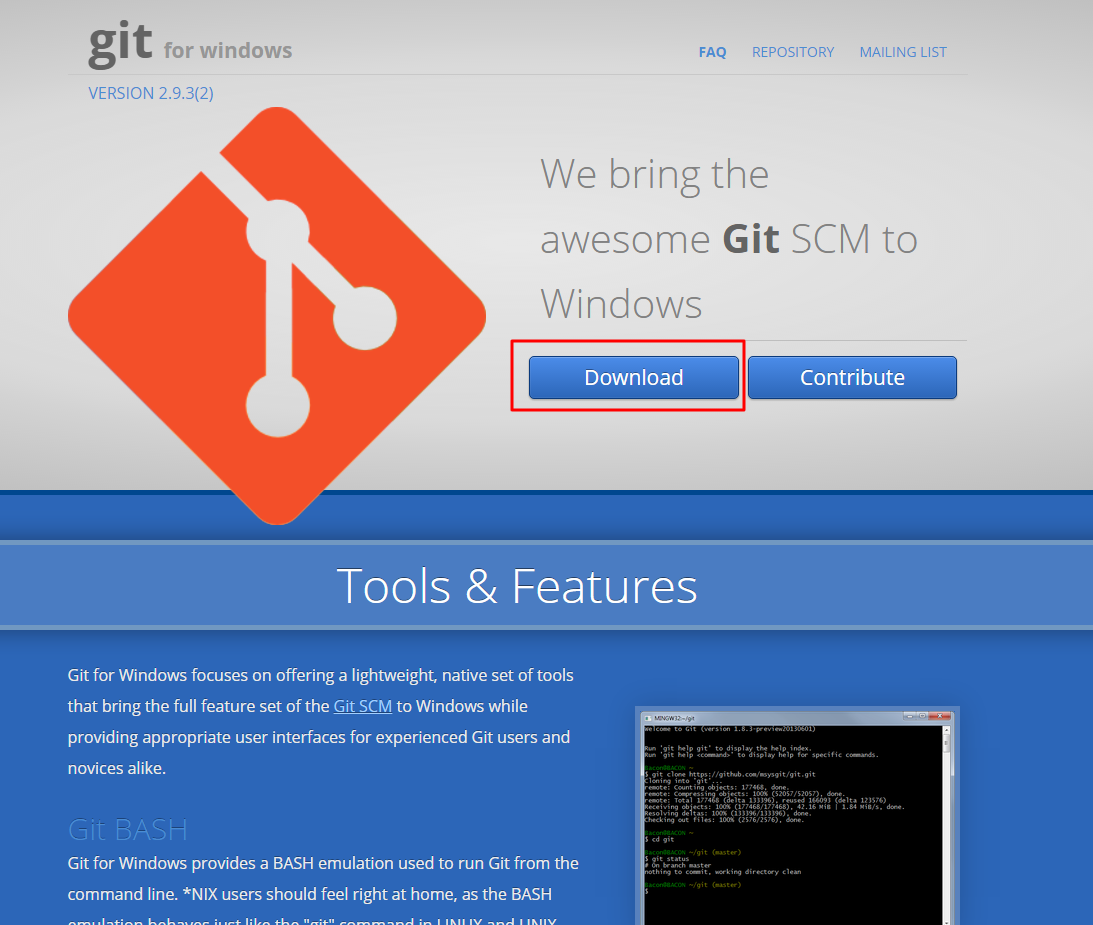
* + 1. **Git Bash [ CLI ] – 선택 사항**

Eclipse plug-in은 로그조회, Git 제어 등의 측면에서 CLI 환경보다는 다소 제한적입니다. Git의 기능을 조금 더 추가적으로 사용하고자 하시는 분들은 Windows Git CLI 환경을 제공하는 Git Bash를 설치하시길 권장합니다.

* “Git For windows(msysGit)”를 설치하시면 자동으로 Git Bash가 설치 됩니다.

**다운로드 링크**

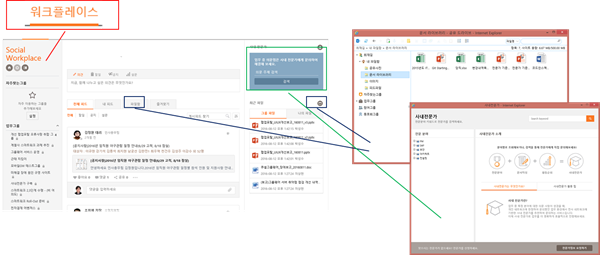
<http://msysgit.github.io/>

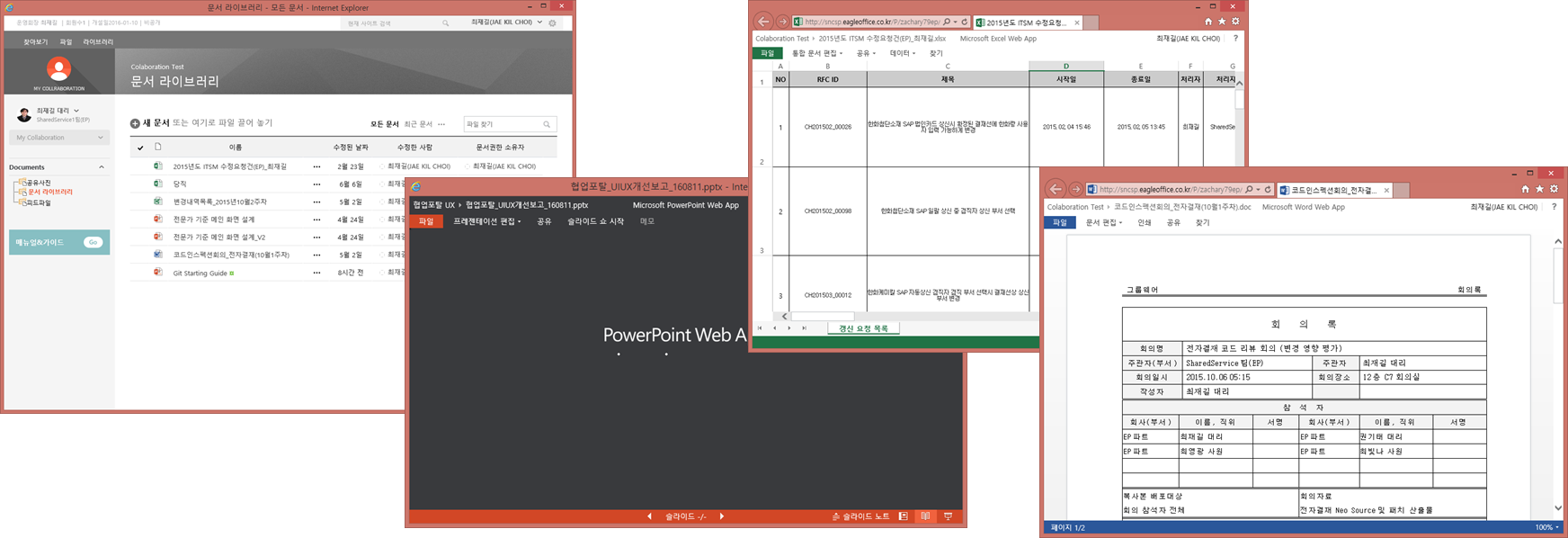


해당 URL에 접속하면 위와 같은 웹 페이지가 보여집니다. 이 페이지에서 별도의 설정 없이 다운로드 하시고 ‘.exe’파일을 설치하시면 됩니다.

* 1. **문서 기록 및 커뮤니케이션 방식 [ 이슈 관리 ]**

본 프로젝트의 이슈 트랙킹 및 문서 공유는 협업포탈의 워크플레이스를 활용 합니다.

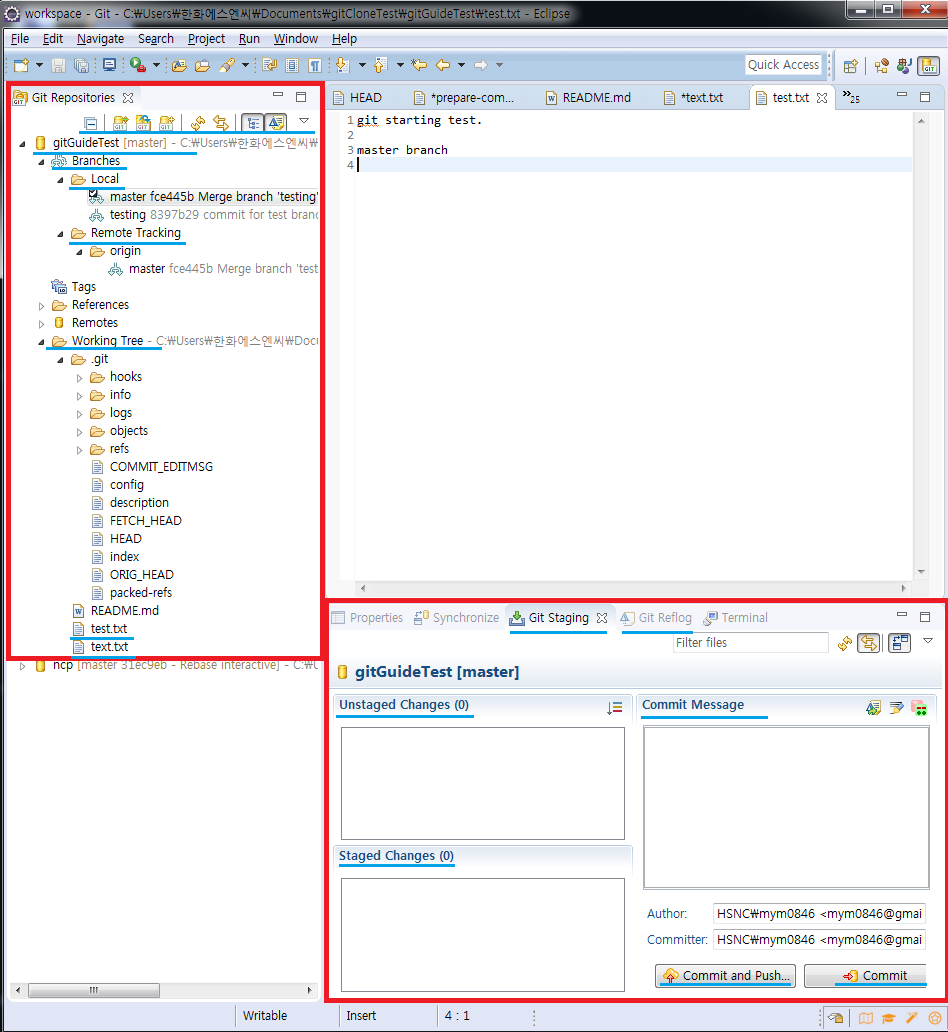




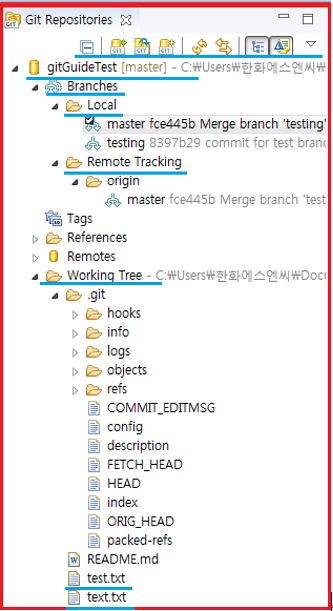
1. **Eclipse Git Plug-in 매뉴얼**

**매뉴얼 설명은 테스트용 원격 저장소를 활용하여 진행됩니다. [ gitGuideTest ]**

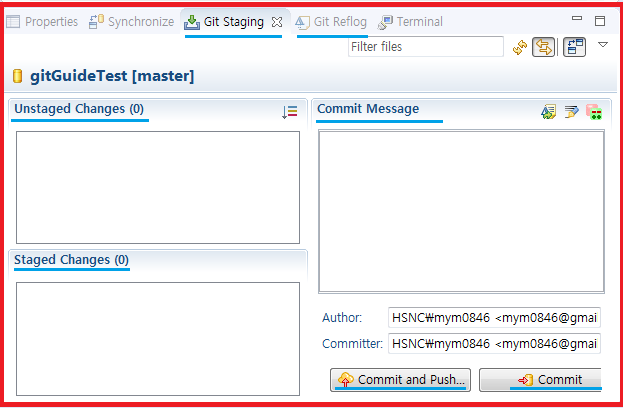
* 1. **User Interface**

****

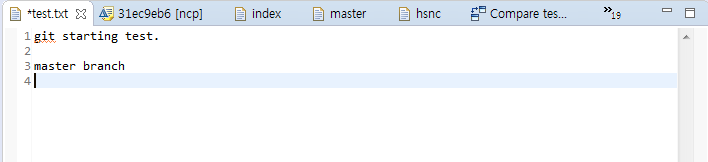
원격 저장소를 로컬에 복제한 후 Eclipse에 프로젝트를 Import한 화면 이미지 입니다. 위 화면 이미지를 분할하여 숙지해야 할 부분들을 설명할 것입니다.

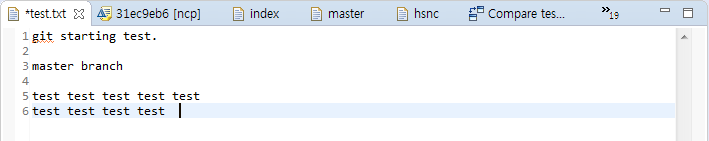


* 상단에 보이는 **Icon**들은 저장소 복제, 저장소 임포트, 저장소 생성, 저장소 상태 새로고침 등 저장소와 관련된 기능들 입니다.
* **‘gitGuideTest’** 는 Git 저장소를 뜻하는 디렉토리입니다.
* **‘Branches’** 는 로컬과 원격 저장소의 브랜치 정보에 대한 목록입니다.
* **‘Working Tree’** 는 Git 디렉토리에서 실질적으로 작업이 이루어지는 공간을 뜻합니다. 예하에 있는 **‘text.txt’ , ‘test.txt’** 작업 파일 또한 **‘Working Tree’** 디렉토리에 존재합니다.

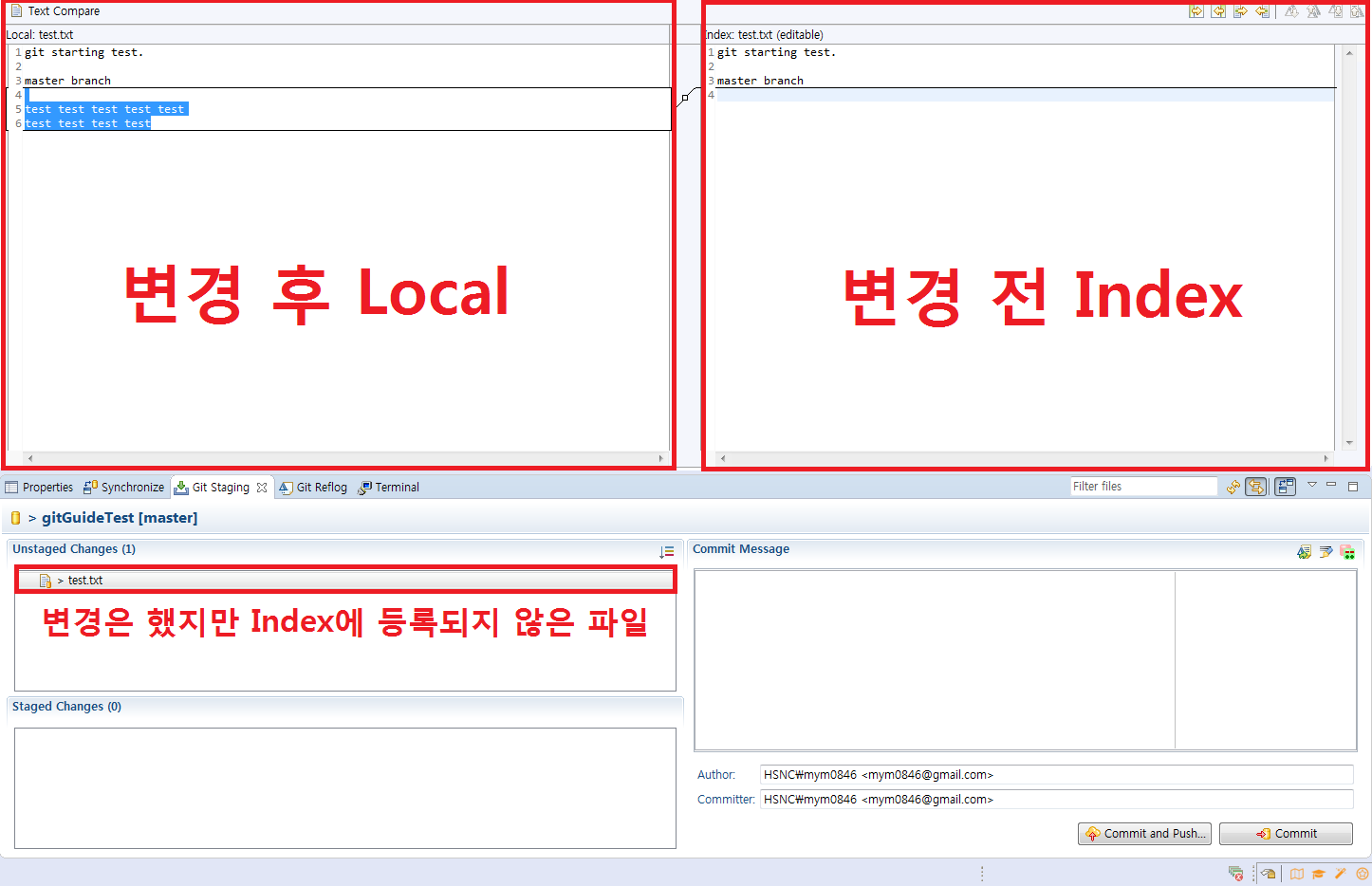


* 상단의 탭 중 **‘Git Staging’**은 작업파일의 변경이력을 관리하는 탭 입니다.
* **‘Unstaged Changes’** 는 작업은 했지만 아직 인덱스에 등록하지 않은 파일 목록
* **‘Staged Changes’** 는 작업한 파일을 인덱스에 등록한 파일 목록
* **‘Commit Message’** 는 작업 후 변경이력을 등록할 때 어떤 부분을 변경하였는지 작업자가 코멘트를 입력하는 부분
* **‘Git Reflog’** 탭은 Git의 작업 변경 이력을 확인할 수 있는 탭 입니다.
* **‘Commit and Push’** 는 변경이력 등록과 원격 저장소와 작업 병합을 한번에 진행하는 기능 입니다.
  1. **Commit**
* 워킹 디렉토리 안에 있는 test.txt 파일을 오픈하여 수정합니다.

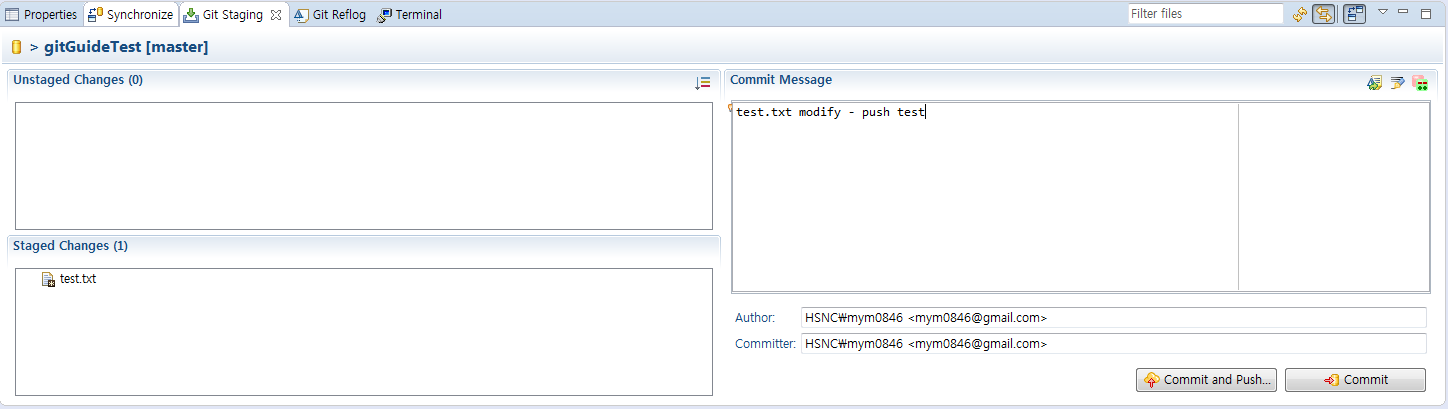




* 수정 후 저장을 하게 되면 다음과 같은 상태창을 확인 할 수 있습니다.

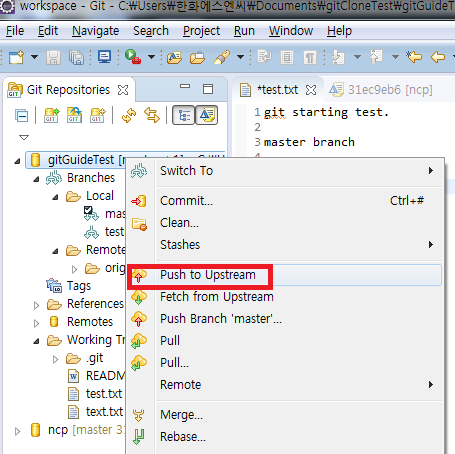


* Unstaged Changes에서 파일을 Drag & Drop 해서 Staged Changes로 옮기고, Commit Message를 입력한 후 Commit 합니다.

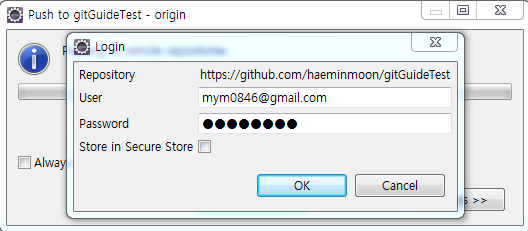


**작업 내용이 인덱스에 등록되고, 인덱스에 등록된 test.txt의 변경 이력을 등록하는 과정**

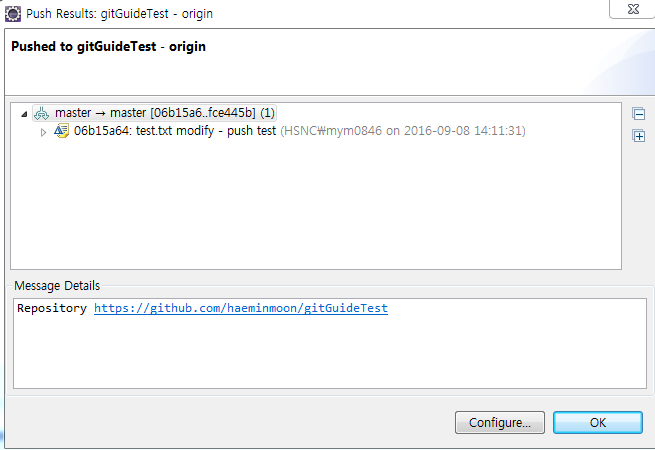
* 1. **Push**
* Commit 완료 후 **‘gitGuideTest’** 를 우클릭 한 후 **‘Push upstream’**을 클릭합니다.



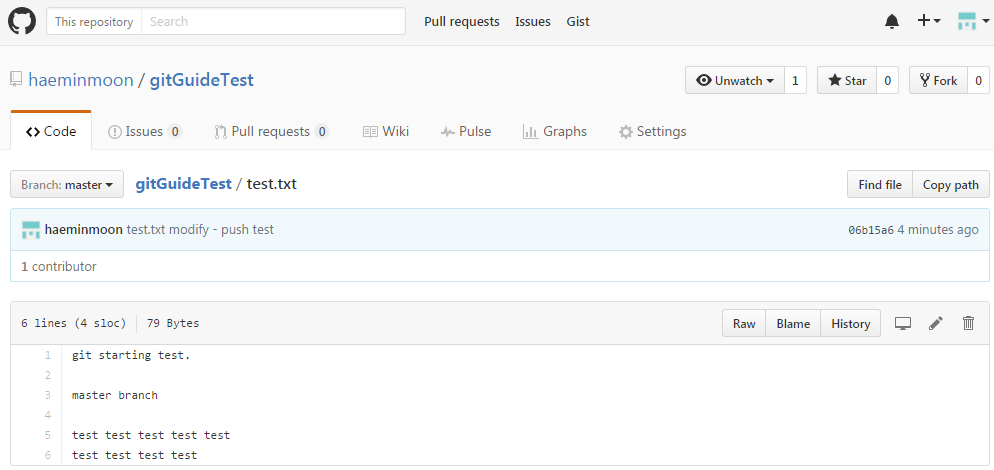
* 다음과 같은 창에서 자신의 Git 계정 정보를 입력합니다.



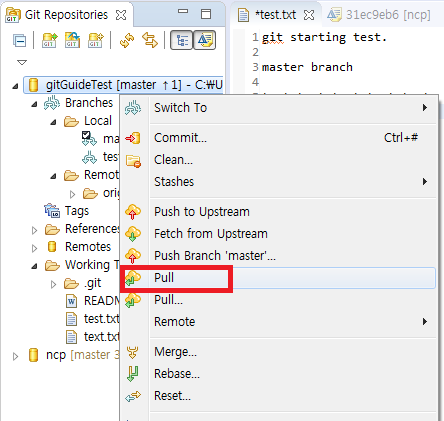
* 다음과 같이 작업 Branch, Commit 내용을 확인 한 후 OK를 누릅니다.



위 과정이 성공적으로 수행되면 원격 저장소에 자신이 작업한 파일이 반영된 것을 확인 하실 수 있습니다.

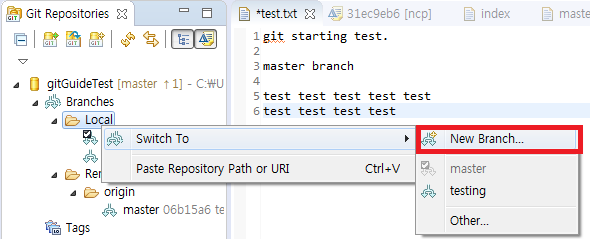


* 1. **Pull**

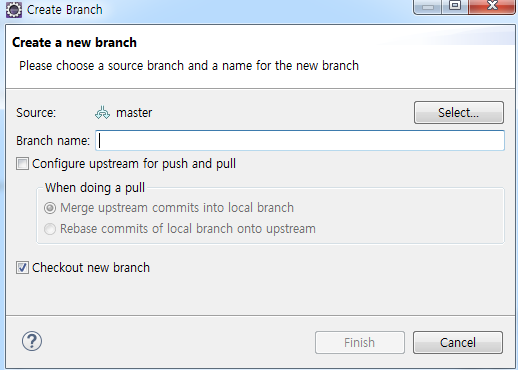


**‘gitGuideTest’** 디렉토리에서 우클릭 후 Pull을 눌러주면, 다른 개발자들에 의해 업데이트된 원격 저장소의 변경 사항이 자신의 로컬 저장소에 병합됩니다.

* 1. **Branch**
     1. **Branch 생성**
* Branch 목록에서 우 클릭 후 ‘New Branch’ 클릭



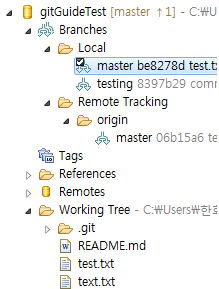
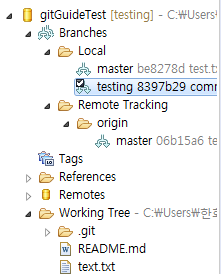
* Branch 명을 기입 후 Finish 클릭



* + 1. **Branch 이동**

Branch 이동이란, Branch를 Checkout 한다는 의미입니다. 즉, 작업 영역을 변경한다는 의미로 해석하시면 됩니다.

* **Branch Checkout 방법은 이동하고자 하는 Branch를 더블 클릭 하시면 됩니다.**

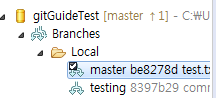
 

위 이미지에서 Branch의 개념을 잘 설명해주고 있습니다. Master에서는 워킹 디렉토리에 ‘test.txt, text.txt’ 파일이 두 개 존재하지만, Testing에서는 ‘text.txt’만 존재합니다. 이는 Branch마다 워킹 디렉토리 자체가 다르다는 것 입니다. 이러한 특징을 이용해서 작업을 모듈별로 분류할 수 있습니다. 이는 분리와 병합을 이용하여 체계적인 분담 작업을 가능하게끔 합니다.

* + 1. **Branch 병합**

현재 로컬 저장소에는 **‘Master, Testing’** 두 개의 Branch가 존재하고 있습니다. 이 Branch들은 원격 저장소에 각각 반영할 수도 있지만, 하나의 Branch에 병합하여 원격 저장소에 반영할 수도 있습니다. 즉, 로컬 영역에서 자신의 작업 영역을 Branch로 모듈화하여 작업하고 나중에 한 번에 Push를 할 수도 있다는 것 입니다.

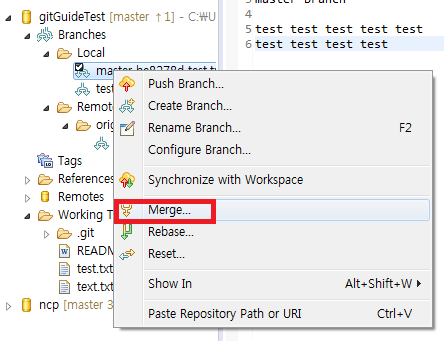
* 우선 ‘master’를 더블 클릭하여 Checkout을 합니다.



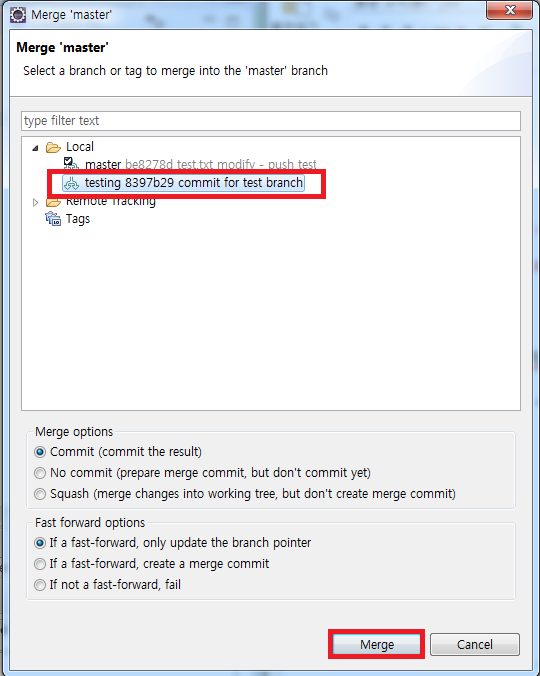
**Branch를 병합 할 시, 기준 Branch와 대상 Branch를 지정해서 기준 Branch로 Checkout해야 합니다.**

**Ex ] (기준 Branch : master) + (대상 Branch : testing) = (병합된 Branch : master)**

* 기준 Branch인 master를 우 클릭 후 Merge를 클릭합니다.



* 대상 Branch인 testing을 선택하고 Merge를 클릭하면 병합 작업이 완료됩니다.



1. **스마트 워크 2, 3단계 프로젝트 버전관리 규약**

버전관리 규약은 Git 충돌 문제, 형상관리 등 관리자와 개발자들 모두에게 발생할 수 있는 문제점들을 미연에 방지하여 작업을 순조롭게 진행하기 위한 목적이니 반드시 숙지해주시기 바랍니다.

* 1. **작업 시작**

**Order :** 작업을 시작할 때는 항상 Pull을 클릭하여 원격 저장소에 최신으로 반영된 내용을 워킹 디렉토리에 복제한(병합) 후 작업을 시작합니다.

**Reason :** 다른 개발자가 자신이 작업하지 않는 동안에 Push를 했을 수도 있기 때문에 충돌을 방지하기 위함 [항시 최신 원격 저장소 내용을 유지해야함 ]

**Order :** 작업 시에는 항상 자신이 담당한 모듈만을 수정합니다. 만약 다른 모듈의 코드를 수정해야 할 일이 발생한다면 해당 모듈 담당자와 반드시 협의 후 진행하시길 바랍니다.

**Reason :** 협업 시 코드관리에 대한 혼동을 야기 할 수 있으며, 이로 인해 향후 충돌 상황이 발생 할 수 있기 때문에 방지하기 위함

* 1. **Commit**

Commit할 때는 어떤 작업을 했는지 ‘Commit Message’에 자세하게 명시해 주시길 바랍니다. 기본적으로 Commit Message는 ‘구현, 수정, 삭제’ 키워드와 함께 명시합니다.

|  |  |
| --- | --- |
| **Keyword** | **Commit Message Sample** |
| 구현 | [구현] 게시판 글 삭제 기능 구현 |
| 수정 | [수정] 검색 알고리즘 효율 개선 (형태소 분석 알고리즘 수정) |
| 삭제 | [삭제] 익명 게시판 서비스 종료 -> 삭제 |

* 1. **Branch**

이번 **‘스마트워크 2, 3단계 롤 아웃 프로젝트의 브랜치 운용’**은 원격 저장소( Repository )의 master Branch를 기준으로 각 계열사 별로 분할하여 운용할 것입니다. 즉, PM or PL들이 일련의 규칙에 따라 Branch명을 명명할 것이며, 직접 운용할 것이므로 Branch 병합 규칙을 잘 지켜주시기 바랍니다.

**Ex) master -> hsnc, hchemical, hdefence ………**

로컬 저장소에서의 Branch 운용은 각자의 Git 사용편의에 맞게끔 운용하셔도 무방합니다. 하지만 **원격 저장소에 작업 내용을 Push할 때는 반드시 ‘master branch’를 기준 Branch로 잡고, Branch들을 병합 후 Push**해 주시길 바랍니다.

**\* master branch 명 변경 금지 \***

* 1. **Push**

작업 내용을 원격 저장소로 Push할 때는 반드시, 일련의 작업이 완료된 후 Push 해주시길 바랍니다. **[ Check list 점검 후, Push ]**

|  |
| --- |
| **Push Check list** |
| 코드 작성이 완료되어야 함 |
| 목표한 모듈의 기능이 완료되어야 함 |
| 작업 완료 후, 컴파일이 성공적으로 가능해야 함 |
| 기능상의 버그가 존재하지 않아야 함 |

* 1. **Conflict [ 장애 상황 발생 시 ]**

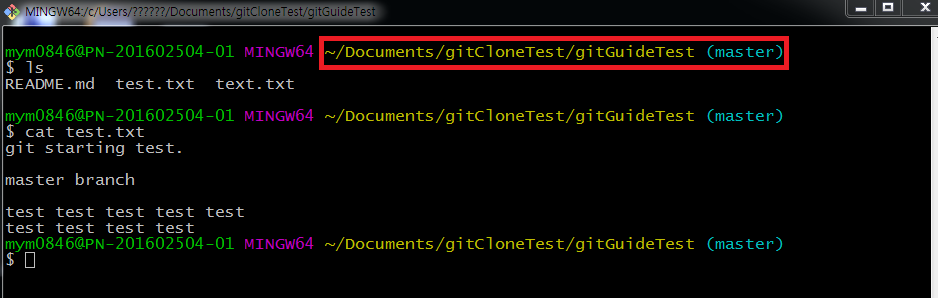
Git에서 발생하는 대부분의 장애는 병합하는 과정에서 충돌 문제로 발생하는 장애입니다. Git 사용자들이 가장 많이 볼 수 있는 에러 메시지가 바로 **“오류가 발생하여 Push or Merge가 거부(Rejected) 당했다”**는 메시지 입니다. 이번 Chapter에서는 가장 빈번하게 발생하는 충돌 문제 상황과 조치법을 설명합니다.

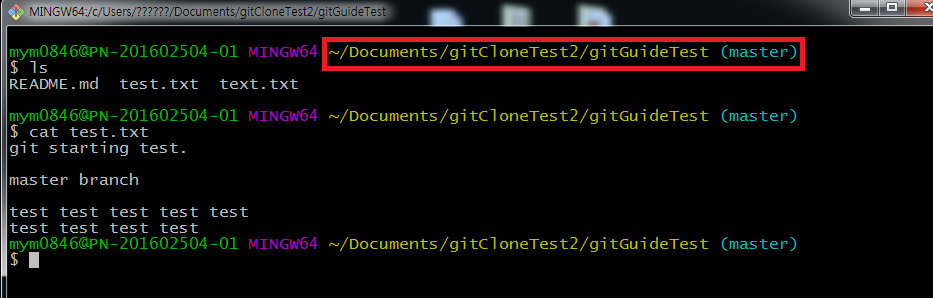
* + 1. **같은 부분을 수정했을 경우**

Git에서 병합 과정이 이루어질 때 자동으로 병합되는 ‘fast merge’ 개념이 있습니다. 이때 코드상에 문제가 없으면 자동 병합이 진행되지만, 만약 두 명의 개발자가 코드의 같은 부분을 수정하였다면 충돌이 발생하게 됩니다.

이 경우 수동으로 병합을 진행해야 합니다. 이 상황을 간단한 예제로 설명하겠습니다.

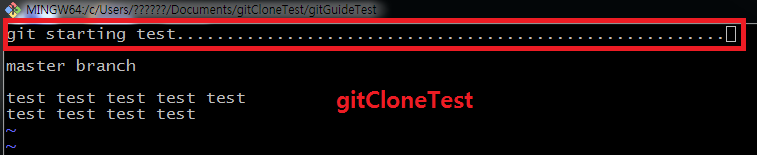
* 두 개의 로컬 저장소를 준비합니다. **( gitCloneTest, gitCloneTest2 )**

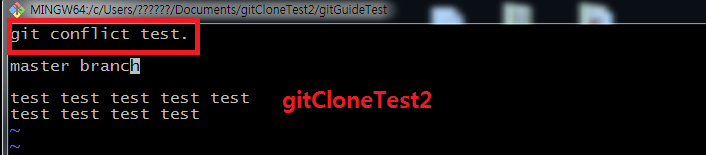




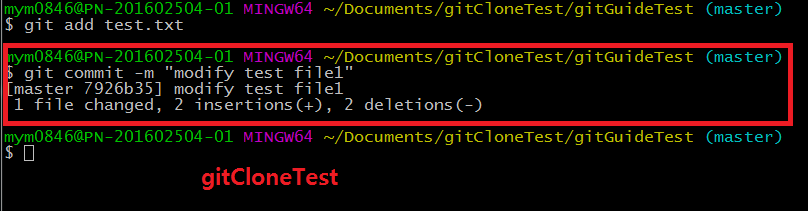
두 로컬 저장소 모두 원격 저장소로부터 Clone 했기 때문에 현재는 동일한 상태입니다.

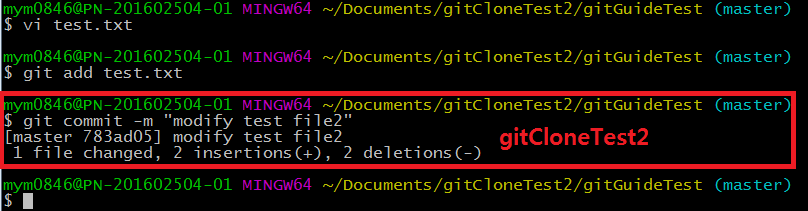
* Vi 편집기를 이용하여 각 로컬 저장소의 test.txt 파일을 다음과 같이 수정합니다.



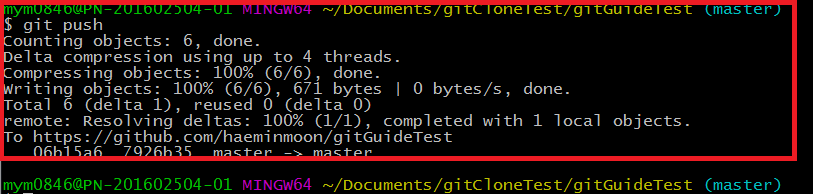


* 파일 수정을 완료한 후 각각 Commit 해줍니다.





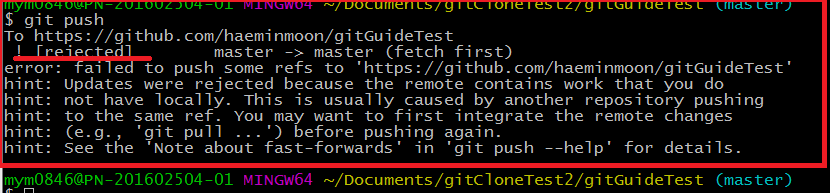
* 먼저 gitCloneTest의 Commit을 먼저 Push합니다.



* 원격 저장소에는 gitCloneTest의 Commit이 제대로 반영되어 있습니다.

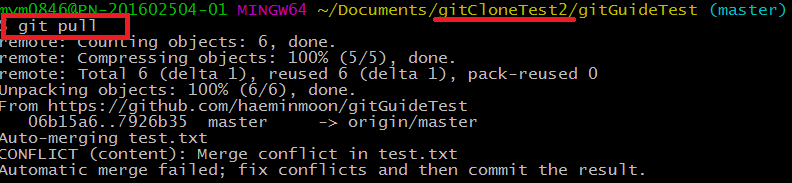


* gitCloneTest2의 Push를 진행합니다.

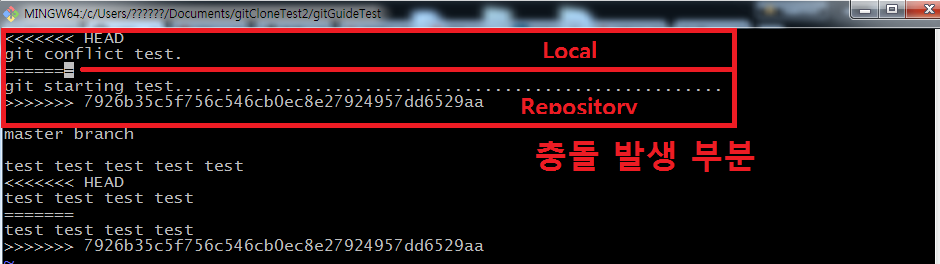


이미지와 같이 Push가 Rejected 되었습니다. 로컬 저장소와 원격 저장소의 Merge가 실패하여 에러 메세지와 함께 Push가 수행되지 않은 것입니다. 이 충돌 문제를 해결하기 위해서는 ‘자동 병합’에 실패 했기 때문에 수동으로 병합해야 합니다.

* gitCloneTest2는 현재 원격 저장소의 최신 버전을 Pull하지 않았기 때문에 원격 저장소의 내용을 로컬 저장소에 Merge해야 합니다. **( git pull을 입력 )**

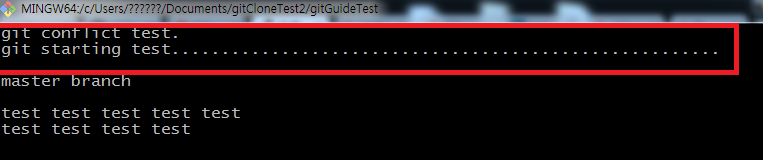


* vi편집기를 이용해서 gitCloneTest2의 test.txt 파일을 확인합니다.

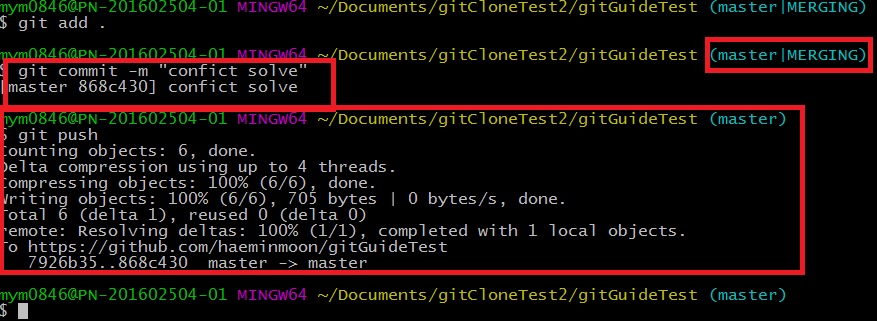


다음과 같이 어떤 라인에서 충돌이 발생했는지, 어떤 코드가 겹쳐서 충돌이 발생했는지 로컬과 원격 저장소로 구분하여 명시 되어있습니다.

* 충돌이 발생한 부분을 다음과 같이 수동으로 변경합니다.



* 수정이 완료 된 후, Commit과 Push를 다시 진행합니다.



* 원격 저장소에 내용이 정상적으로 반영된 것을 확인 하실 수 있습니다.



* + 1. **최신 원격 저장소의 내용을 반영하지 않았을 경우**

Commit과 Branch에서 병합 충돌 문제가 발생했을 때 Push를 하는 것은 당연하게 원격 저장소로부터 Rejected 메시지를 받게 됩니다. 하지만 이보다 더 빈번하게 발생 하는 문제는 바로 Pull하지 않은 상태에서 작업 진행 후 Push하는 것입니다.

문제의 발생 원인은 이렇습니다. Push 명령어 내부에는 자동 병합[Merge] 기능이 포함되어 있습니다. 이 기능은 병합을 수행하기 이전에 스냅샷을 비교한 후 병합 기능을 수행합니다. 하지만 원격 저장소의 최신 코드를 Pull하지 않았기 때문에 원격 저장소와 워킹 디렉토리의 변경 작업 외의 코드들이 일치 하지 않았기 때문입니다. [스냅샷 비교]

이를 해결하기 위해서는 Push 전에 Pull이나 Fetch로 원격 저장소의 최신 코드를 로컬 저장소에 병합한 후 Push 해야 합니다.

1. **학습자료 링크**
   1. **누구나 쉽게 이해할 수 있는 Git 입문**

<https://backlogtool.com/git-guide/kr/>

* 1. **Pro Git**

<https://git-scm.com/doc>

* 1. **Egit( Eclipse Git plug-in ) Document**

<http://www.eclipse.org/egit/documentation/>

* 1. **SVN 능력자를 위한 Git 개념 가이드**

http://www.slideshare.net/einsub/svn-git-17386752

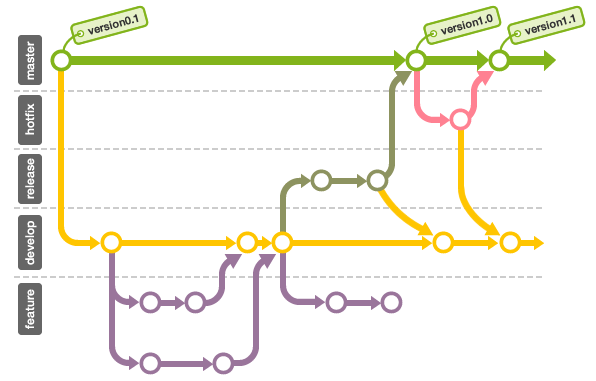
1. **별첨**
   1. **Git Flow [ Branch Work Flow ]**

**“Branch”를** 잘 운용한다면 더욱 완벽한 형상관리가 가능하게 됩니다. 그래서 현재도 Git 컨트리뷰터들은 유지보수, 충돌사항 대처, 작업관리 등 완벽한 CI를 위해 많은 연구를 진행하고 있다고 합니다. 이에 대해 **“A successful Git branching model”** 이라는 성공적인 Branch 운용 모델에 대해 간단하게 소개를 진행 할 것입니다.

[**http://nvie.com/posts/a-successful-git-branching-model/**](http://nvie.com/posts/a-successful-git-branching-model/)

이 운용 모델에서는 Branch를 크게 4가지 종류로 나눠 개발을 진행합니다.

* **메인 브랜치(Main branch)**
* **피처 브랜치(Feature branch) 또는 토픽 브랜치(Topic branch)**
* **릴리스 브랜치(Release branch)**
* **핫픽스 브랜치(Hotfix branch)**



## **메인 브랜치(Main branch)**

**'master branch'**와 **'develop branch'**, 이 두 종류의 Branch를 보통 Main branch로 사용합니다.

* **master**  
  **'master branch'** 에서는, 배포 가능한 상태만을 관리합니다. Commit할 때에는 태그를 사용하여 배포 번호를 기록합니다.
* **develop**  
  **'develop branch'** 는 앞서 설명한 통합 Branch의 역할을 하며, 평소에는 이 Branch를 기반으로 개발을 진행합니다.

## **피처 브랜치(Feature branch)**

이 Branch는 새로운 기능 개발 및 버그 수정이 필요할 때 **'develop branch'** 로부터 분기합니다. **‘feature branch’**에서의 작업은 기본적으로 공유할 필요가 없기 때문에, 원격으로는 관리하지 않습니다. 개발이 완료되면 **'develop branch’**로 병합하여 다른 사람들과 공유합니다.

## **릴리즈 브랜치(Release branch)**

**‘release branch’**에서는 버그를 수정하거나 새로운 기능을 포함한 상태로 모든 기능이 정상적으로 동작하는지 확인합니다. **‘release branch’**의 이름은 관례적으로 Branch 이름 앞에 'release-' 를 붙입니다. 이 때, 다음 번 Release를 위한 개발 작업은 **'develop branch'**에서 계속 진행해 나가면 됩니다.

**‘release branch’**에서는 Release를 위한 최종적인 버그 수정 등의 개발을 수행합니다. 모든 준비를 마치고 배포 가능한 상태가 되면 **'master branch'**로 병합시키고, 병합한 Commit에 Release 번호 태그를 추가합니다.

**‘release branch’**에서 기능을 점검하며 발견한 버그 수정 사항은 **'develop branch'** 에도 적용해 주어야 합니다. 그러므로 배포 완료 후 **'develop branch'**에 대해서도 병합 작업을 수행합니다.

## **핫픽스 브랜치(Hotfix branch)**

배포한 버전에 긴급하게 수정을 해야 할 필요가 있을 경우, **'master branch'** 에서 분기하는 Branch입니다. 관례적으로 Branch 이름 앞에 'hotfix-'를 붙입니다.

예를 들어 **'develop branch'** 에서 개발을 한창 진행하고 있는 도중에 이전에 배포한 소스코드에 아주 큰 버그가 발견되는 경우를 생각해 보세요. 문제가 되는 부분을 빠르게 수정해서 안정적으로 다시 배포해야 하는 상황입니다. **'develop branch'**에서 문제가 되는 부분을 수정하여 배포 가능한 버전을 만들기에는 시간도 많이 소요되고 안정성을 보장하기도 어렵습니다. 그렇기 때문에 바로 배포가 가능한 **'master branch'**에서 직접 Branch를 만들어 필요한 부분 만을 수정한 후 다시 **'master branch'**에 병합하여 이를 배포하려고 하는 것이죠.

이 때 만든 **‘hotfix branch’**에서의 변경 사항은 **'develop branch'**에도 병합하여 문제가 되는 부분을 처리해 주어야 합니다.