소프트웨어공학

GitHub 보고서

- Title : GitHub & Git strategy -

Class	**소프트웨어공학, Class 01**
Professor	**이찬근**
Team number	**Team 18**
Team Members	**김찬중 (20213780)** **박민용 (20213113)** **엄태형 (20202029)** **이정우 (20202021)** **태아카 (20234483)**
Author	**김찬중**

목차

1. Introduction

A) GitHub 및 Git 사용 방안

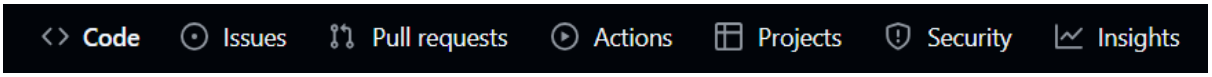
- 프로젝트 주소: https://github.com/BigBeautifulM/SETermProject18
- GitHub 및 Git 사용 경험이 적은 팀원을 위한 사용 방안 및 문제 해결 방안을 설명

A – 1 - 1) GitHub & Git 사용 방법: Local에 Workspace 생성

- 생성된 GitHub 저장소를 Local에 clone하여 사용해야 한다.
- 우리의 프로젝트를 예를 들어 설명한다. local에서 작업할 폴더를 생성 후 terminal을 켜준다.
- 'git clone "https://github.com/BigBeautifulM/SETermProject18"'을 입력
- 로컬에 Workspace가 생성된다.

A – 1 - 2) GitHub & Git 사용 방법: Pull request 관리 및 에러 핸들링

- 로컬에서 master branch가 아닌 자기가 만든 branch인 경우 push 이후 master branch와의 merge가 필요하다.



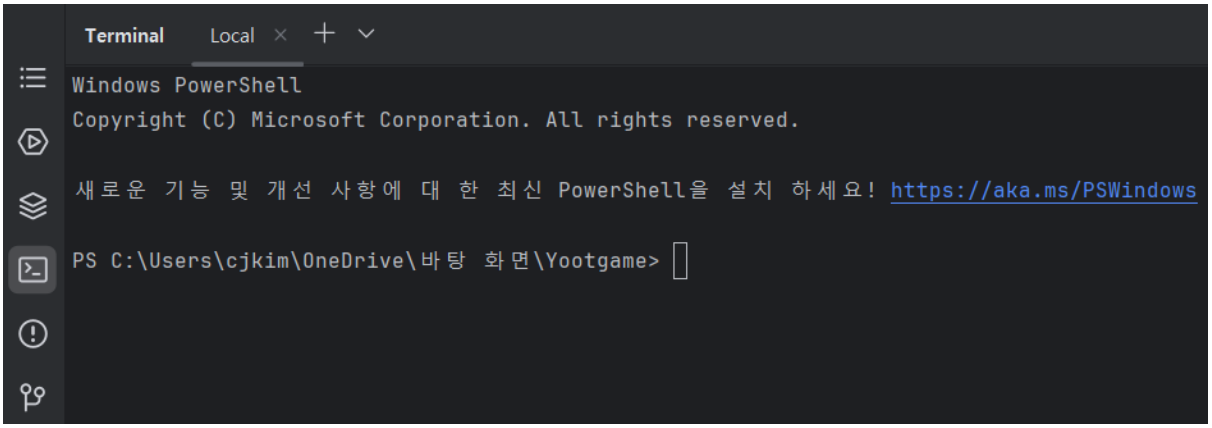
- GitHub의 Repository를 들어가면 상단에 'Pull requests'라는 옵션이 보일 것이다.

- 위와 같이 New pull request를 실행하여 merge를 진행하면 된다.
- 보통의 경우 local에서 upstream push를 하게 될 경우 repository를 접속하자 마자 pull request 제안이 보이게 된다.
- Merge를 할 때 Conflict 메시지가 뜨면서 병합이 안되는 경우가 있다.
- 이는 Master branch에서의 변경 사항이 병합하려는 branch에 적용되지 않았기 때문이다.
- 현재 GitHub는 온라인 상에서 Conflict 코드 부분을 알려주고 GitHub 내에서 코드를 수정함으로써 병합을 할 수 있게 도와준다.

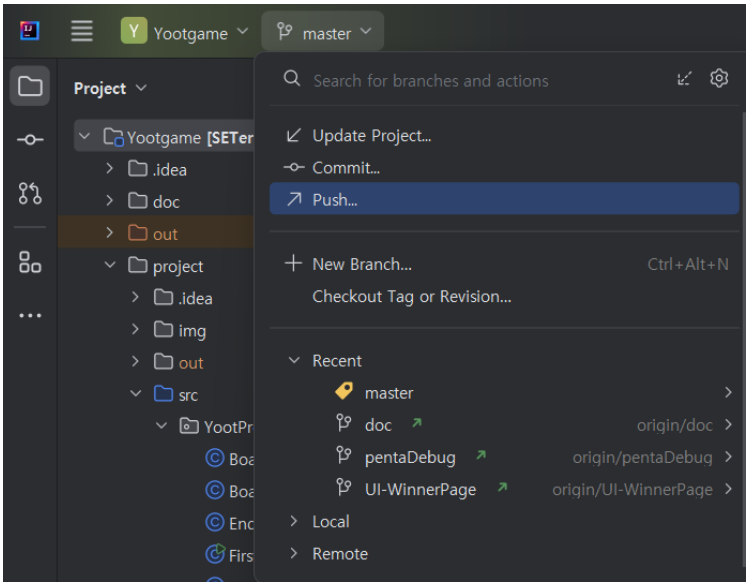
A – 1 - 3) GitHub & Git 사용 방법: Conflict의 사전 예방 방지

- Master branch가 갱신된다면 항상 'git pull origin master'을 통해 최신화를 진행해야 한다.
- Master branch를 pull하고 현재 작업중인 branch와의 병합도 필요하다.
- Git checkout [브랜치 이름] 혹은 git switch [브랜치 이름]를 이용해 작업 중인 branch로 분기한다. 이후 git merge master을 통해 master에 적용된 최신 사항을 작업중인 branch로 병합해야 한다.
- 만약 다른 사람이 작성한 remote branch를 다른 팀원이 수정하고자 한다면 git pull -b [브랜치 이름] origin/[브랜치 이름]을 통해 해당 branch만 local로 pull하여 작업을 진행해야한다.

A – 2) Intelij dev 환경에서의 효율적인 Git 사용 방안



- Intelij 하단 좌측의 Terminal icon을 누르면 위와 같이 Intelij 내에서 terminal을 조작할 수 있다.
- 실시간으로 업데이트 사항을 조원과 소통하며 branch 관리 및 병합 작업을 수행하면 능률을 높일 수 있다.
- Windows의 경우 기본적으로 PowerShell로 연결되며 CMD, Linux Kernel도 연결하여 프로그래밍을 진행할 수 있다.
- 그 밖에도 Setting-Plug in에서 Git, GitHub 관련 기본 setting을 조절하여 프로젝트를 진행하는 데에 도움을 받을 수 있다.



- 일반적인 Terminal과 다르게 Intelij는 상단의 directory 선택창의 우측을 보면 git 및 github관련 명령어 집합이 버튼으로 mapping되어 있다.
- Commit 로그를 바로 작성할 수 있으며 Branch 생성, Push, Branch 제거도 가능하다.
- Intelij의 프로젝트의 경우 가끔 git pull을 통해 최신화를 했더라도 갱신 사항이 바로 적용되지가 않는 경우가 있다. 이때 위의 Update Project를 통해 해결할 수 있다.

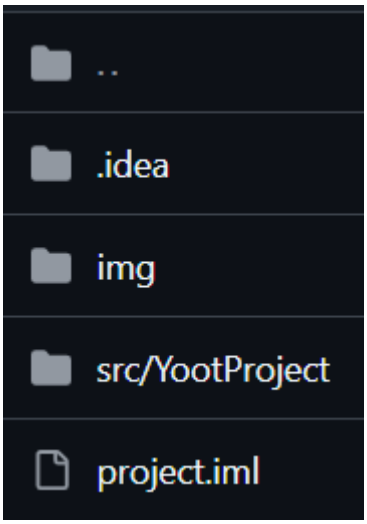
2. Body

B) GitHub

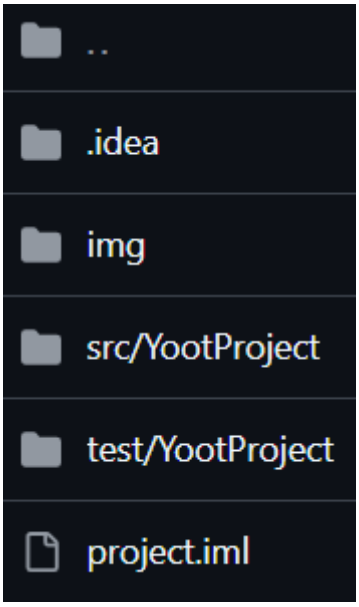
B – 1) Directory 구조

- 프로젝트 폴더의 주소의 기본 골조이다. Doc 폴더는 문서와 관련된 모든 내용이 있고 project 폴더에는 java swing으로 진행한 윷놀이 프로젝트 코드가 있다.

■ Doc 폴더의 하위 폴더 구조이다. Diagram, 테스트 시나리오, 개요, 회의 내용 및 발표문 그리고 2차 과제 제출 때 사용할 문서를 나누어 저장한다.



■ 이는 project 하위 폴더 구조이다. Img directory에서 게임 판에 사용되는 이미지를 상대주소로 불러와 사용한다. Src directory는 우리가 게임 상에서 돌아가는 UI, Logic에 대한 코드의 집합 폴더이다. YootProject Package 상에서 작업했기에 directory 이름이 src/YootProject이다.



- Test branch로 분기하면 다음과 같이 project 하위에 test/YootProject directory가 추가 되어있다. 이는 YootProject package를 참조하는 junit test code가 있는 것을 의미한다.

B – 2) Merge 및 commit 관리

- 팀원들의 GitHub 사용 경험을 고려하여 GitHub 사용 규칙 및 Commit 규칙을 정하였다.
- Merge를 할 때에는 '김찬중' 학생을 참고인으로 Pull Request를 생성한다. 생성 시 Discord와 같은 연락 도구를 사용하여 Pull Request 요청 사실을 모든 조원에게 전송한다.
- 디버깅이 완료되고 문제가 없을 시 master branch에 병합을 진행한다.
- Commit의 경우 message 작성에 대해 규칙을 정하였다.
- Commit log는 다른 작성자가 알아볼 수 있도록 유의미한 문장으로 작성되어야 한다. 예를 들어 프로젝트 내의 육각형 모양의 판의 UI를 수정한 경우 '육각형 판 UI 수정'과 같이 다른 팀원들이 한 눈에 알아볼 수 있도록 수정사항을 메시지에 드러내야 한다.
- Commit은 사소한 부분이 생기더라도 message로 남겨야 한다. Commit history를 통해 에러가 발생한 부분을 예측할 수 있고 복구를 진행할 수 있기 때문이다.

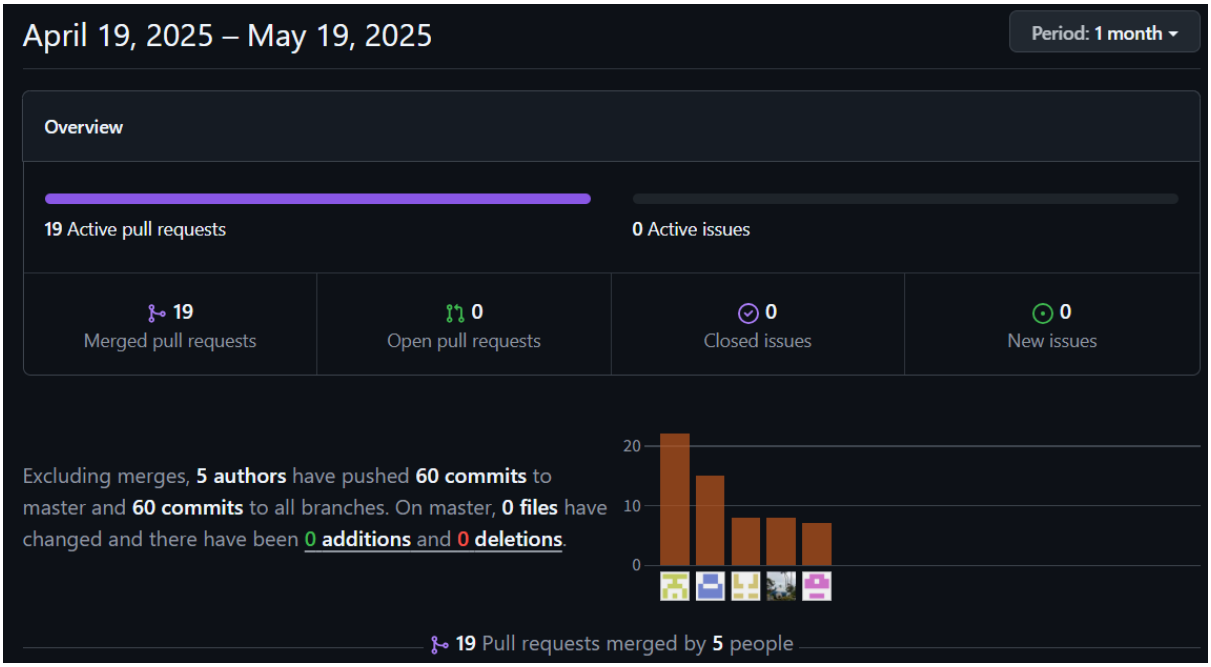
- Project를 진행하면서 주로 발생하였던 commit의 내용은 주로 Additions였으며 Deletions의 경우 Debug를 통해 테스트 코드 혹은 필요 없는 변수를 줄이는 용도로 사용되었다.

C) Git

C – 1) Git Branch 전략

- 우리의 project의 역할 분담은 UI, Logic, 기획 & Document로 크게 세 부분으로 나눌 수 있다. 모든 프로젝트의 진행사항을 master branch 상에서만 진행한다면 자신의 진행한 부분을 다른 사람이 잘못 수정하거나, 자신의 프로젝트 수행 정도를 잘 인식하지 못하는 경우가 있다.
- 따라서 각 인원별로 branch를 생성하여 프로젝트를 진행한다.
- UI의 경우 자신이 맡은 UI 부분에 대한 이름을 이용하여 branch를 생성한다.

- Branch의 이름을 결정할 때에는 [수행 파트][개발 or 디버그]와 같이 자신이 맡은 역할 중 어느 부분을 진행한지의 여부를 파악할 수 있게 해야 한다.
- Logic 부분의 경우 branch 이름을 결정할 때 [logic 이름][개발 or 디버그]와 같은 형태로 최대한 branch 생성을 유도하였다.



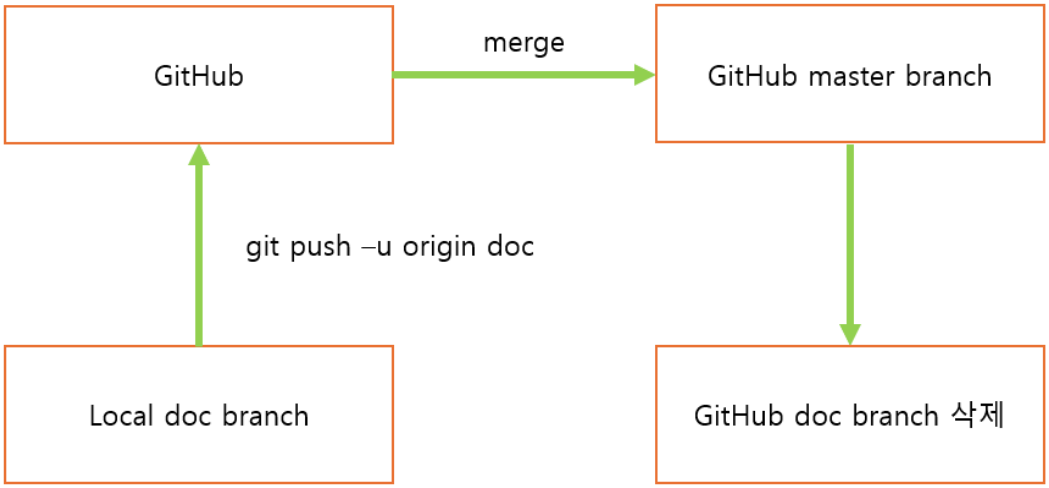
■ 보고서 작성 직전 동안 약 19회의 Pull Request가 있었다. 제일 많았던 pull request의 경우 로직에서 논리적 좌표를 실제 좌표와 대조하는 부분의 에러가 많아 Debug 횟수가 많았다. 이와 같이 Debug를 여러 번 진행할 때 하나의 branch에서 모든 것을 진행하는 것보단 각 Debug 부분을 소분할 하여 Merge Request를 진행함으로써 Conflict 가능성을 최대한 낮추었다.

■ 상단의 사진은 실제로 진행한 Branch 합병 및 생성에 대한 work flow 일부이다.

■ Conflict를 최소화하기 위해선 master branch에서 분기하여 자신의 branch를 생성하는 것이다.

■ 위의 경우 한 명의 팀원이 하나의 task를 수행하면 merge를 통해 master에 병합을 진행했다. 이후 다른 branch에서 작업하던 팀원들은 pull request 소식을 통해 master branch를 갱신하였고 local branch에 merge함으로써 Conflict를 회피할 수 있었다.

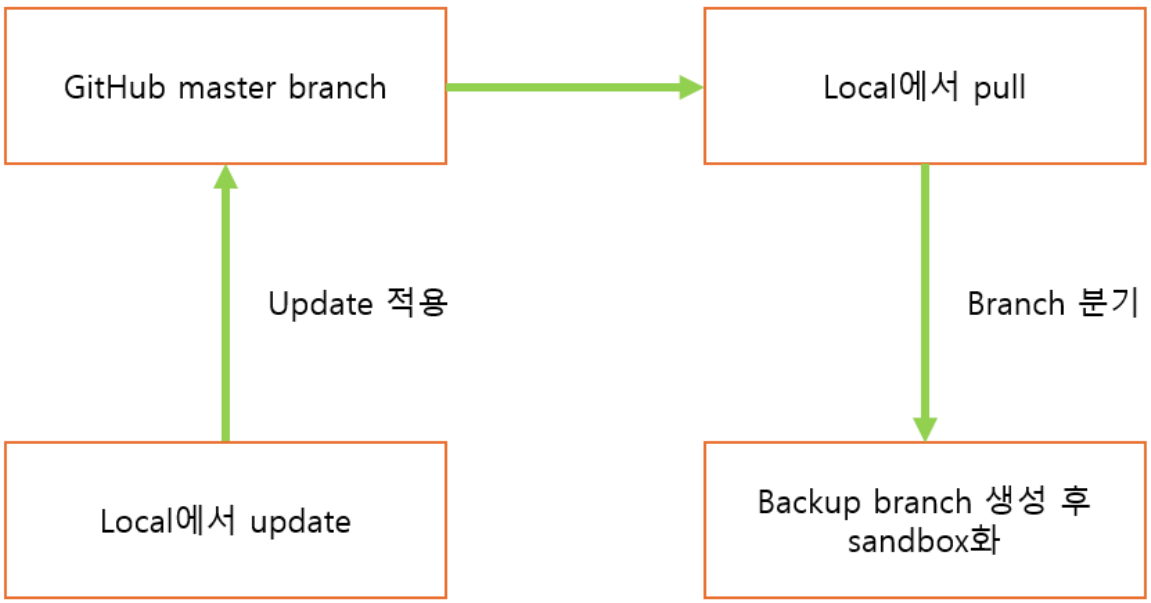
■ 코드 외의 documentation의 경우 doc branch를 통해 관리 감독하였다. 원격 Repository (즉 GitHub)에선 doc을 merge한 후 삭제를 항상 진행하였다. (아래 참고)



■ 'push -u'는 upstream push로 remote branch에 없는 local branch를 push하기 위한 명령이다. 위의 경우 GitHub 내의 branch에 doc branch가 없다는 것이 전제가 되어 진행되는 process인 것을 알 수 있다.

C – 2) Backup 전략

■ 프로젝트를 진행하다 보면 불의의 사고로 인해 데이터가 이상하게 수정되거나 삭제되는 경우가 있다. 이러한 불상사를 미연에 방지하기 위한 규칙이 있다.



■ 위의 사진은 프로젝트에서 backup을 어떻게 진행하였는지를 도식화한 사진이다.

- Local에서 update가 발생하여 remote repository의 변화를 발생시켰다고 가정하자. (이때의 가정은 update가 local 환경에서 완벽하게 돌아가는 경우를 상정한다.) 팀원 중 한 명(김찬중; 기획, GitHub, 문서 담당)이 local에 backup branch를 생성한다. Master branch를 local에도 update를 적용한 후 이를 backup branch에 merge 시킨다. 이후 다음 update까지는 backup branch를 격리시킨다.
- 위와 같은 과정은 다른 팀원이 실수로 master branch에서 작업을 하여 모든 branch와의 conflict가 발생했을 때 hard하게 복구할 수 있게 해준다.
- Git의 사용을 능수능란하게 할 수 있다면 stash나 commit log를 이용한 history backup을 할 수 있지만 팀원의 Git 사용 여부를 고려한 결과 위와 같은 방법이 안전할 것 같다고 생각했다.

3. Conclusion (소감 및 요약)

- Git, GitHub 사용 경험이 생각보다 적었지만 팀원들이 빠르게 전략을 적용하고 Discord, Notion 등과 같은 communication app을 통해 소통을 자유자재로 해주어 Conflict와 같은 큰 에러가 발생하진 않았다.
- 프로젝트가 끝난 후 모든 팀원들이 자신만의 Git branch 전략, Project 유지보수 전략에 대해 많은 이해를 가져갈 것 같다.
- Git commit log에서 가장 많이 발생했던 key word는 디버깅이었다. 코드 작성과 비교했을 때 약 2~3배가량의 Debug commit이 있었다. 또한 branch에 대한 병합 요청 시 많은 부분이 logic과 UI 사이의 conflict를 해소하기 위한 디버깅이었다.
- 프로젝트 진행을 통해 코딩보다 디버깅이 오래 걸린다는 것을 몸소 느낄 수 있었고 unit test와 같이 소규모의 test가 얼마나 중요한 지를 잘 알 수 있었다.