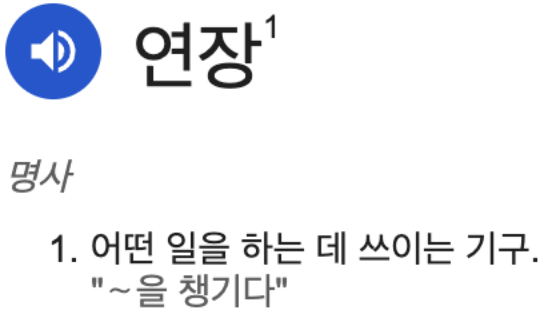
실용주의 프로그래머

3. 기본 도구

1. 도구

도구는 사용자의 생각을 완성된 제품으로 끊임 없이 전달하는 통로와도 같다.

=> 도구는 손의 연장이라 할 수 있음



+) 일반 텍스트의 힘

지식을 일반 텍스트로 저장하면 아래 일들이 가능해진다.

1. 지원 중단 시 보험
2. 기존 도구의 활용
3. 더 쉬운 테스트

1. 도구

셸

GUI 환경의 기능은 일반적으로 설계자가 의도한 범위를 넘어설 수 없다.

하지만 개발자는 자주 그 모델 이상이 필요함. 특히 자동화에 있어서 자주 필요하게 됨

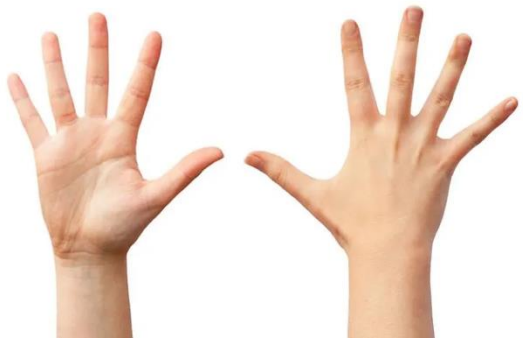
따라서 셸을 잘사용하면 생산성이 급상승할 수 있게 됨

```
grep '^import ' *.java | \
sed -e's\.*import *//' =e's/;.*$//' | \
sort -u > list
```

1. import로 시작하는 줄을 찾음
2. 결과에 대해서 한 줄 씩 진행(sed)
3. 's/._import _//' import로 시작하는 모든 텍스트에 대해서
4. s/;.₩*$//: ;이후부터 라인의 끝까지 없애는 역할을 함
5. sort -u > list : 정렬하면서 -u 중복 제거해서 결과를 list라는 파일에 저장

1. 도구

파워 에디팅



=> 도구는 손의 연장이라 할 수 있음

손을 쓸 때 어떻게 써야겠다고 생각을 하고 쓰지는 않음
도구도 이렇게 써야 생산성이 극대화될 수 있지 않을까

개발자에게 손 같은 존재란? => IDE

1. 도구

파워 에디팅

자주 쓰는 VSCode 단축키 공유해보기

1. cmd + P – 파일 이름으로 검색
2. cmd + F - 파일 내부에서 검색
3. cmd + shift + F - 전체 워크스페이스에서 검색
4. cmd + 클릭 (F12) - 함수 정의로 이동
5. cmd + D – 선택영역과 동일한 영역 다중 선택
6. F2 - 선언부 이름변경 + 모든 사용처 적용
7. cmd + / - 주석처리
8. option + 방향키 – 단어단위로 커서 이동

더 잘 쓰기 위한 방안

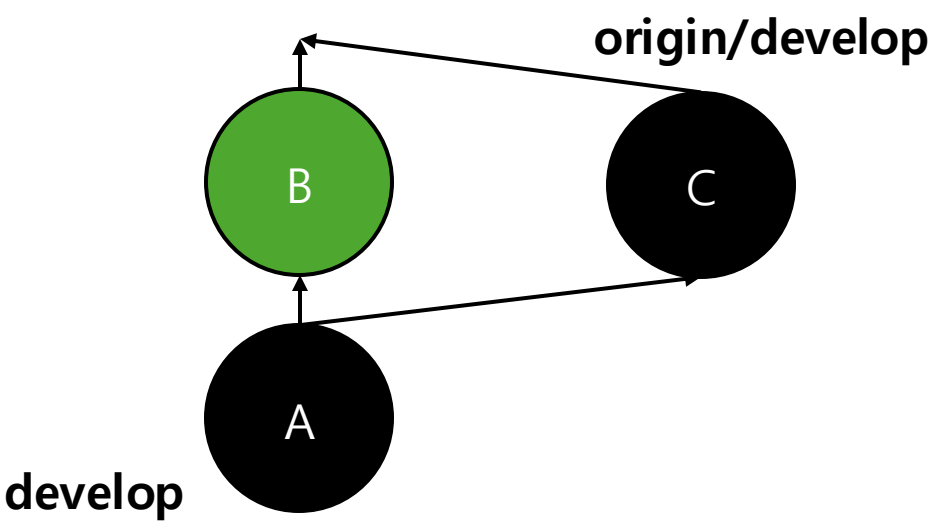
1. 자신이 에디터를 사용하는 모습을 관찰
2. 같은 일을 반복하는 것을 발견할 때
3. 더 나은 방법을 찾아서 적용하기

1. 도구

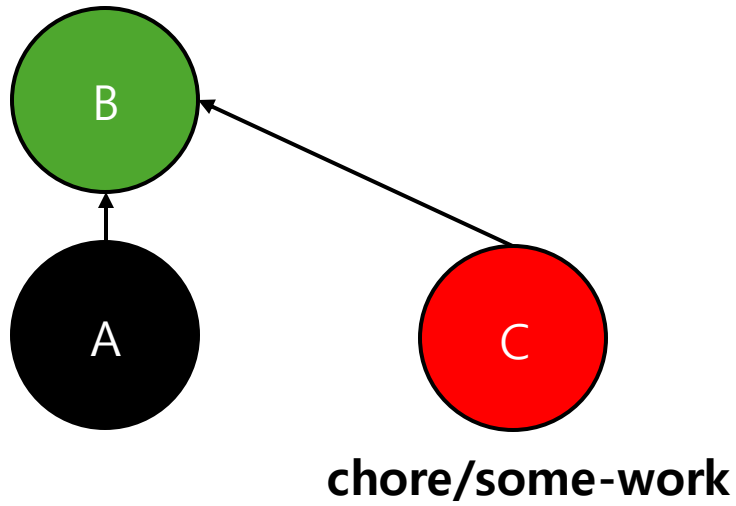
버전관리

버전 관리 시스템은 거대한 실행 취소 키와 같음

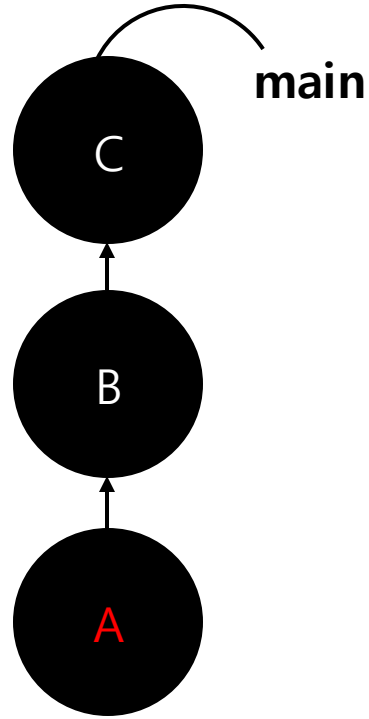
1.



2.



3.



1. 도구

이쯤에서 돌아보는 git

1. git reset 과 git revert
- reset 브랜치의 HEAD를 이동시키는 방식으로 해당 시점으로 이동하는 방식
- revert는 브랜치 HEAD를 이동시키지 않고 특정 commit을 제거하는 새로운 커밋을 추가하는 방식.

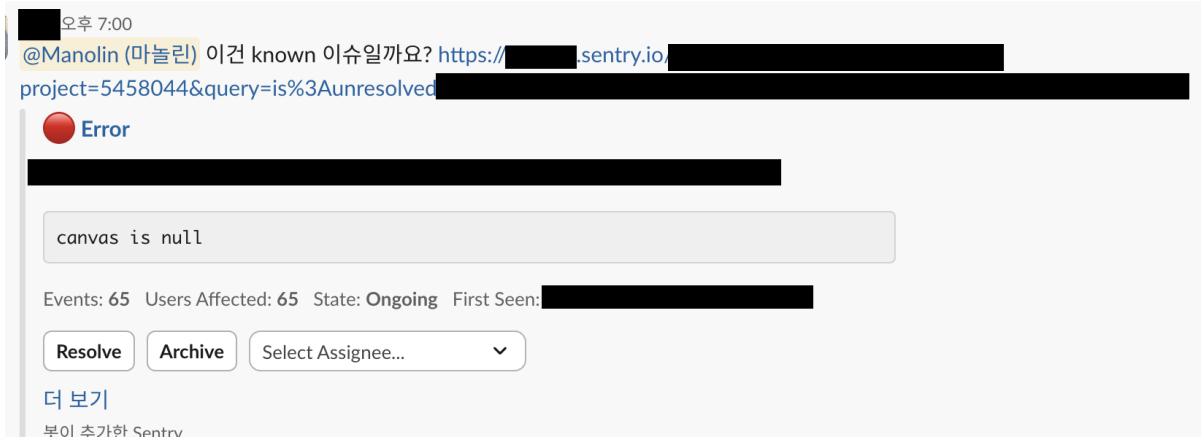
2. git cherry-pick
- A브랜치에서 B브랜치에 있는 해시값을 가져와서 해당 변경점을 적용시켜버릴 수 있음

3. git rebase
- 현재 브랜치 커밋을 다른 브랜치 최신 커밋 위로 재배치
이전 커밋들이 추가되므로, 당연히 해시값이 바뀜~

2. 디버깅

디버깅 제 1원칙

당황하지 말 것



근시안의 함정에서 벗어나라.
표면에 보이는 증상만 고쳐내려는 욕구를 이겨내야 한다.

2. 디버깅

실마리 찾기

1. 버그 보고한 유저를 인터뷰
2. 로그 수집
3. 인공적인 테스트 필요

디버깅 전략

1. 버그 재현하기 -> 버그 픽스의 첫걸음은 재현임
2. 오류 메시지 좀 읽기

개인적으로 사용하는 전략

1. 입력 값이나 상황에 대해 경계값에 대해 바꿔가면서 검증
2. 가설 세우고 소거법으로 디버깅

4. 실용주의 편집증

1. 완벽한 소프트웨어/정확한 소프트웨어

완벽한 소프트웨어

완벽한 소프트웨어는 존재하지 않음
실용주의 프로그래머는 자기 자신 역시 믿지 않음.
자기 자신도 완벽한 코드를 작성할 수 없음을 알기에 실수에 대비한 방어책을 마련함

정확한 소프트웨어 (프로그램)

정확한 프로그램이란, 자신이 하는 일이라고 주장하는 것보다 많지도 적지도 않게
딱 그만큼만 하는 프로그램을 말함

2. 계약에 의한 설계 (DBC)

선행조건

호출되기 위해 참이어야 하는 것
루틴의 요구사항이며, 위반되면 루틴이 호출되면 안됨

후행조건

루틴이 자기가 할 일이라고 보장하는 것

불변조건

클래스나 객체가 유지해야 하는 항상 참인 조건

ex) 계좌 잔액은 절대로 음수가 될 수 없다.

이후 아래 문법을 써서 구현

1. **assert**
2. **If 문**
3. **decoration**

2. 계약에 의한 설계 (DBC)

굳이 DBC를 사용해서 구현하지 않더라도 아래 요소들을 생각해보면 더 나은 소프트웨어를 작성할 수 있게 됨

1. 유효한 입력 범위는 무엇인가?
2. 경계 조건은 무엇인가?
3. 루틴이 뭘 전달한다고 약속하는가?/약속하지 않는가?

3. 죽은 프로그램은 거짓말을 하지 않는다.

있을 수 없는 일이 발생했을 때, 상황을 부정하지 말 것

그 일은 발생 했다!

일단 그 놈의 오류 메시지를 읽을 것

그리고 그 일이 일어났다는 사실을 우리는 알아야 한다.

1. Switch case 에 default케이스가 필요한 이유
2. catch할 것
   catch할 때 도 마찬가지로 응집성 있게 관리할 것

3. 죽은 프로그램은 거짓말을 하지 않는다.

말도 안되는, 상식에 반하는 일은 생각보다 많다.

1. 1752년 9월은 19일 밖에 없다.
2. 비유클리드 기하학에서 삼각형 내각의 합은 180도가 아닐 수 있다.
3. 윤초 때문에 1분은 61초일 수 있다.
4. 언어에 따라 overflow가 발생해서 +1을 할때 부호가 음수로 변할 수도 있다.

망치지 말고 멈춰라

오류가 발생하는 경우, 빠르게 더이상 프로그램이 진행되지 않게 해서 해를 끼치지 못하게 할 필요가 있다.

4. 리소스 사용의 균형

지역적으로 행동하라

리소스 할당/해제는 지역적으로,
작은 단위에서 자신이 시작한 것을 자신이 끝낼 필요가 있음

exception 이 나는 경우는 finally로 처리할 수 있음

중첩 할당

리소스 할당 순서의 역순으로 해제하기
여러 번 동일한 구성의 리소스를 할당하는 경우 언제나 같은 순서로 할당할 것 -> 데드락 방지

5. 헤드라이트를 앞서가지 말 것

너무 큰 작업을 하지 말 것

작은 단계들을 밟아라. 언제나
더 진행하기전에 피드백을 확인하고 조정하라
너무 큰 단계나 작업은 하지 않게 될 것임

너무 큰 단계는 **예언**과도 같음
블랙스완이 기다리고 있을 수 있음

끝