54p. ~ 65p.

▼ 부수 효과를 일으키지 마라!

> 📢 부수 효과는 거짓말이다. 함수에서 한 가지를 하겠다고 약속하고선 남몰래 다른 짓도 하니까. (54p)

먼저 아래의 코드를 한번 천천히 읽어봅시다!

```
public class UserValidator {
    private Cryptographer cryptographer;

    public boolean checkPassword(String userName, String password) {
        User user = UserGateway.findByName(userName);
        if (user != User.NULL) {
            String codedPhrase = user.getPhraseEncodedByPassword();
            String phrase = cryptographer.decrypt(codedPhrase, password);
            if ("Valid Password".equals(phrase)) {
                Session.initialize();
                return true;
            }
        }
        return false;
    }
}
```

위에 코드는 부수 효과를 일으키고 있습니다. 다들 어디인지 눈치 채셨나요?

부수 효과를 일으키는 부분은 Session.initialize() 부분입니다!

checkPassword라는 함수 이름만으로 세션이 초기화된다는 사실이 드러나지 않습니다. 이러한 부수효과 때문에 checkPassword라는 함수는 세션을 초기화 해도 괜찮은 경우에만 호출이 가능합니다. 이 결함을 시간적인 결합이라고 합니다.

시간적인 결함은 혼란을 일으킵니다. 만약에 부수 효과로 숨겨진 경우에는 더더욱 혼란이 가중되게 됩니다. 이를 방지하기 위해 함수 이름에 분명히 명시하는 편이 좋습니다.

▼ 명령과 조회를 분리하라!

> 📢 함수는 뭔가를 수행하거나 뭔가에 답하거나 둘 중 하나만 해야 한다.(56p)

다음 함수를 살펴봅시다.

```
public boolean set(String attribute, String value);
```

이 함수는 이름이 attribute인 속성을 찾아 값을 value로 설정한 후 성공여부를 반환하는 함수입니다. 보기에는 별 이상이 없어 보이는데 이 함수를 가지고 코드를 쓰면 아래와 같이 됩니다.

```
if (set("username", "unclebob"))...
```

위에 코드를 보다보면 다음과 같은 가능성들이 열리게 됩니다.

1. “username”이 “unclebob”으로 설정되어 있는지 동일 여부 반환
2. “username”을 “unclebob”으로 설정 후에 성공 여부 반환

함수를 구현한 개발자는 set을 동사로 의도했지만 if문과 함께 보면 형용사로 느껴지고 그래서 의도와 다르게 “username 속성이 unclebob으로 설정되어 있다면”으로 읽히게 됩니다. 이를 해결하기 위해서는 명령과 조회를 분리해야 합니다.

```
if (attributeExists("username")) {          //조회
  setAttribute("username", "unclebob");           //명령
}
```

▼ 오류 코드보다 예외를 사용하라!

> 📢 명령 함수에서 오류 코드를 반환하는 방식은 명령/조회 분리 규칙을 미묘하게 위반한다.(57p)

아래의 코드를 보자!

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configkeys.deleteKey(page.name.makeKey()) == E_OK) {
      logger.log("page deleted");
    } else {
      logger.log("configKey not deleted");
    }
  } else {
    logger.log("deleteReference from registry failed");
  }
} else {
  logger.log("delete failed");
  return E_ERROR;
}
```

첫번째 줄에 deletePage(page) == E_OK 부분은 앞서 말한 동사/형용사 혼란을 일으키지는 않지만 여러 단계로 중첩되는 코드를 야기합니다. 오류 코드를 반환하면 호출자는 오류 코드를 곧바로 처리해야 하기 때문입니다.

반면 오류 코드 대신 예외를 사용하명 오류 처리 코드가 원래 코드에서 분리되어 코드가 깔끔해집니다.

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
} catch (Exception e) {
  logger.log(e.getMessage());
}
```

try/catch 블록은 혼란을 일으키기 때문에 별도 함수로 뽑아내는 편이 좋습니다.

```
private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

여기서 주의해야하는 점은 오류 처리 역시 한 가지 작업만 해야 된다는 점입니다.

▼ 반복하지 마라!

> 📢 어쩌면 중복은 소프트웨어에서 모든 악의 근원이다.(60p)

중복은 크게 3가지 문제를 야기한다.

1. 코드의 길이를 늘린다.
2. 알고리즘이 변하면 여러 곳을 손봐야 한다.
3. 수정 중에 한 곳이라도 빠뜨리면 오류가 발생한다.

```
bool Dataloader::load_kitti_calibration(
  const std::string& calibration_path,
  kitti::Calibration& calibration_data)
{
  std::ifstream ifs(calibration_path);
  std::string line;
  double value;
  if (!ifs.is_open())
  {
    std::cout << "cannot open calib_path : " << calibration_path << std::endl;
  }

  std::getline(ifs, line);
  std::stringstream ss(line.substr(4));
  for (int row = 0; row < 3; ++row)
  {
    for (int col = 0; col < 4; ++col)
    {
      ss >> value;
      calibration_data.P0(row, col) = value;
    }
  }

  std::getline(ifs, line);
  ss.clear();
  ss.str(line.substr(4));
  for (int row = 0; row < 3; ++row)
  {
    for (int col = 0; col < 4; ++col)
    {
      ss >> value;
      calibration_data.P1(row, col) = value;
    }
  }

  std::getline(ifs, line);
  ss.clear();
  ss.str(line.substr(4));
```

```
  for (int row = 0; row < 3; ++row)
  {
    for (int col = 0; col < 4; ++col)
    {
      ss >> value;
      calibration_data.P2(row, col) = value;
    }
  }

  std::getline(ifs, line);
  ss.clear();
  ss.str(line.substr(4));
  for (int row = 0; row < 3; ++row)
  {
    for (int col = 0; col < 4; ++col)
    {
      ss >> value;
      calibration_data.P3(row, col) = value;
    }
  }

  std::getline(ifs, line);
  ss.clear();
  ss.str(line.substr(9));
  for (int row = 0; row < 3; ++row)
  {
    for (int col = 0; col < 3; ++col)
    {
      ss >> value;
      calibration_data.R0_rect(row, col) = value;
    }
  }

  std::getline(ifs, line);
  ss.clear();
  ss.str(line.substr(16));
  for (int row = 0; row < 3; ++row)
  {
    for (int col = 0; col < 4; ++col)
    {
      ss >> value;
      calibration_data.velo_to_cam(row, col) = value;
    }
  }

  std::getline(ifs, line);
  ss.clear();
  ss.str(line.substr(16));
  for (int row = 0; row < 3; ++row)
  {
    for (int col = 0; col < 4; ++col)
    {
      ss >> value;
      calibration_data.imu_to_velo(row, col) = value;
    }
  }
  ifs.close();
```

```
  return 0;
}
```

▼ 구조적 프로그래밍

> 📢 구조적 프로그래밍의 목표와 규율을 공감하지만 함수가 작다면 위 규칙은 별 이익을 제공하지 못한다.(61p)

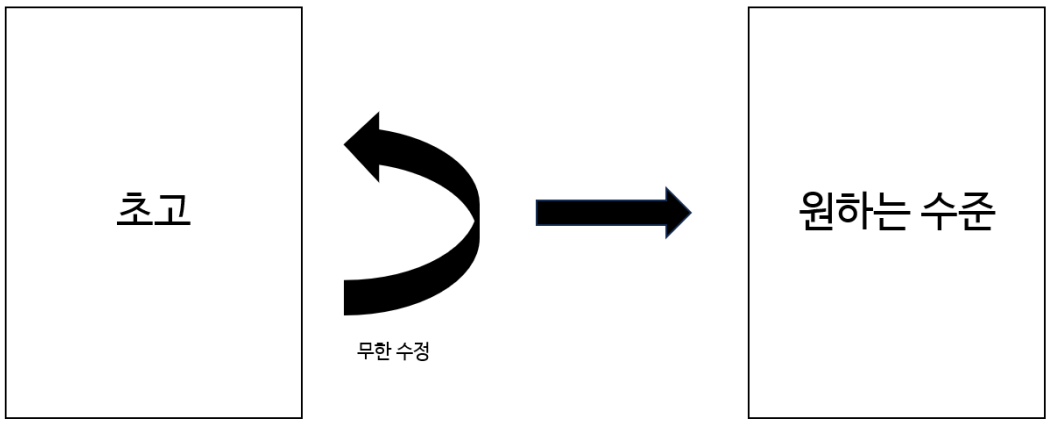
- 구조적 프로그래밍 : 모든 함수와 함수 내 모든 블록에 입구와 출구가 하나만 존재해야 한다.

즉 return문이 하나여야 한다.

함수를 작게 만든다면 return, break, continue를 여러 차례 사용해도 괜찮다. 오히려 단일 입/출구 규칙보다 의도를 표현하기 쉬워진다.

▼ 함수를 어떻게 짜죠?

> 📢 소프트웨어를 짜는 행위는 여느 글짓기와 비슷하다.(61p)



항상 단위테스트는 통과!

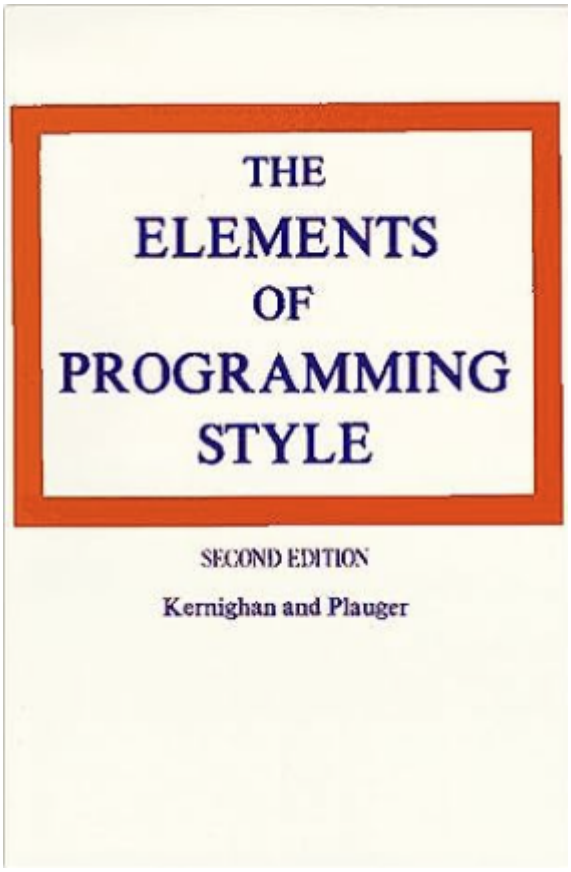
처음부터 탁 짜내지 못한다. 그게 가능한 사람은 없다.

▼ 결론

> 📢 하지만 진짜 목표는 시스템이라는 이야기를 풀어가는 데 있다는 사실을 명심하기 바란다. 여러분이 작성하는 함수가 분명하고 정확한 언어로 깔끔하게 같이 맞아떨어져야 이야기를 풀어가기가 쉬워진다는 사실을 기억하기 바란다.(62p)

3장은 함수를 잘 만드는 기교를 소개했다. 여기서 설명한 규칙을 따른다면 길이가 짧고, 이름이 좋고, 체계가 잡힌 함수가 나오리라.

▼ 참고문헌




실용주의 프로그래머
20주년 기념판
20th Anniversary Edition
Design Patterns
Elements of Reusable Object-Oriented Software
GoF의 디자인 패턴
개정판
재사용성을 지닌 객체지향 소프트웨어의 핵심 요소
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Protec Media


AGILE SOFTWARE DEVELOPMENT
로버트 C. 마틴
소프트웨어
개발의 지혜
원칙, 디자인 패턴, 실천방법