클린코드 스터디

오류 처리, 경계

오류 처리

오류 처리

- 잘못될 가능성은 늘 존재한다.
- 뭔가 잘못되면 바로 잡을 책임은 우리 프로그래머에게 있다.
- 오류 처리로 인해 프로그램 논리를 이해하기 어려워 진다면, 클린 코드라 부르기 어렵다.

오류 처리

- 오류 코드보다 예외를 사용하라
- Try-Catch-Finally 문부터 작성하라
- 미확인 예외(Unchecked Exceptions)를 사용하라
- 예외에 의미를 제공하라
- 호출자를 고려해 예외 클래스를 정의하라
- 정상 흐름을 정의하라
- null을 반환하지 마라
- null을 전달하지 마라

오류 코드보다 예외를 사용

```
// Bad
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Check the state of the device
    if (handle != DeviceHandle.INVALID) {
      // Save the device status to the record field
      retrieveDeviceRecord(handle);
      // If not suspended, shut down
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended. Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

```
// Good
public class DeviceController {
  ...
  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }

  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);
    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
  }
  ...
}
```

Try-Catch-Finally부터

- 강제로 예외를 일으키는 테스트 케이스를 작성하라.
- try-catch-finally문을 작성한다.
- try-catch 구조로 범위를 정의하고 논리를 추가한다.

Unchecked Exceptions를 사용하라

- Checked Exception -> 반드시 예외를 처리해야 함.
- Checked Exception은 장점 대비 지불해야 하는 비용이 큼.
    - 어떤 메소드가 checked exception throw시킬 수 있다면,
    - catch하는 caller전 단계의 모든 메소드에 해당 exception을 정의해야 한다.

Unchecked Exceptions를 사용하라

- Open/Closed Principle 위배
  - **개방-폐쇄 원칙**(OCP, Open-Closed Principle)은 '소프트웨어 개체(클래스, 모듈, 함수 등등)는 확장에 대해 열려 있어야 하고, 수정에 대해서는 닫혀 있어야 한다.'는 프로그래밍 원칙이다.

  - 상위레벨 메소드에서 하위레벨 메소드 디테일을 알아야 한다.
    -> 캡슐화 깨짐

예외에 의미를 제공하라

- 전후 상황을 덧붙인다. -> 오류 발생 원인과 위치를 찾기 쉽다.
- 오류 메세지에 정보를 담에 함께 던진다.
- 실패한 연산 이름과 실패 유형도 언급한다.

사용에 맞게 Exception class 선언하라

- 써드파티 라이브러리를 사용하는 경우, 그것을 wrapping 하여 사용하자

정상적인 흐름을 정의하라

- catch문에서 예외적인 상황(special case)를 처리해야 할 경우 코드가 더러워진다.

- Special Case Pattern을 활용하자 -> 클래스를 새로 만들거나 객체를 조작해 특수 사례 처리

Null을 반환하지 마라

- Null check는 나쁘다!

- Special Case Pattern을 쓰자.

Null을 전달하지 마라

- Null을 메소드로 넘기는건 더 나쁘다.

- null이 파라미터로 들어오지 못하게 하는 것이 제일 좋다.

경계

경계

- 시스템에 들어가는 모든 소프트웨어를 직접 개발하는 것은 어렵다.
- 외부 라이브러리나, 다른 팀이 제공하는 컴포넌트를 사용하는 경우가 많다.

- 인터페이스 제공자와 인터페이스 사용자 사이에는 특유의 긴장이 존재

경계

```
public class Sensors {
    // 경계의 인터페이스(이 경우에는 Map의 메서드)는 숨겨진다.
    // Map의 인터페이스가 변경되더라도 여파를 최소화할 수 있다. 예를 들어 Generic을 사용하던 직접
    // 이는 또한 사용자의 목적에 딱 맞게 디자인되어 있으므로 이해하기 쉽고 잘못 사용하기 어렵게 된다.

    private Map sensors = new HashMap();

    public Sensor getById(String id) {
        return (Sensor)sensors.get(id);
    }
    //snip
}
```

- Map의 예시
    - Map 사용시 -> 객체 유형을 Sensors 에서 관리하자.

경계

- 학습 테스트를 하자! -> 간단한 테스트 케이스를 통해 외부 코드를 익한다.

- 아직 존재하지 않는 코드 -> 자체적으로 인터페이스 정의해서 설계하자.

- clean 경계

클린코드 중간고사

OX Quiz

- 함수의 크기는 작을 수록 좋다
- TODO 주석은 나쁜 주석이다.
- 잡종 구조는 피하는 편이 좋다.
- 변수 선언부에서, 가로로 나란히 정렬해서 코딩하는 것이 좋다.
- 구조적 프로그래밍의 방법론대로 함수를 구현하는 것이 좋다.
- 학습 테스트는 외부 코드를 익히기 위한 간단한 테스트 케이스이다.

다음을 알맞게 고치시오.

- public static void copyChars(char a1[], char a2[])
- getNameString()
- int dsm;
- bool change = true;
- string address_city;
  string address_home_number;
  string address_post_code;

해답

- public static void copyChars(char source[], char destination[])
- getName()
- int day_since_modification;
- bool is_changed=true;
- class address {
  string city;
  string home_adress;
  string post_code;
  }

빈칸을 채우시오.

- 클래스나 객체가 예외적인 상황을 캡슐화 해서 처리하도록 하는 패턴을 (　　)라 부른다.

- final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
  -> 위와같은 코드를 (　　)라 부른다.

생각해봅시다.

- 클린코드 스터디를 진행하면서 추가하고 싶던 '나만의 클린코드 원칙'?

- 책에서 가장 인상깊던(뼈를 때리는) 조언