Code Complete

- OverEngineering을 주의하자
- No Silver Bullet : 이거 하나면 모든게 다 해결된다고 이야기하지만 그런 기술은 없다
- 좋은 프로그래밍의 원칙
    - DRY (Don't Repeat Yourself) : 중복은 죄악시하는 것. 제일 나쁜게 반복이다 (단순한 코드의 중복이 아닌 중복된 작업, 중복된 기능등을 말한다.)
    - YAGNI (You aren't gonna need it) : 언젠가는 필요할 것이라는 예측은 대부분 맞지 않는다. 필요할 때에 최소한의 설계를 하자.
    - KISS (Keep It Simple, Stupid) : 단순하고 알기 쉽게 디자인하자.
    - 유지보수 하는 사람을 위한 코드를 만들라
    - 최소 놀람의 원칙: 내코드를 보는 다른 사람을 놀라게해서는 안된다 (Coding covention을 잘 따라서 다른 사람을 놀라지 않도록 해야한다)
    - High Cohesion, Loose Coupling (높은 응집도, 낮은 결합도) : 클레스간의 결합은 최소화를하자

- Programming antipatterns (하면 안되는 행동)
    - Reinventing the wheel: 이미 잘 만들어진 것을 다시 만드는 일은 가급적 하지말라
    - Cargo cult programming : 의미나 장단점, 동작방식은 모르고 그냥 따라하기만 하는 것
    - Coding by exception/Exception handling : Exception코드안에 Business Logic이 들어가면 안된다.
    - Avoiding/swallowing exception: 예외를 피하거나 무시하는 것 (try/catch로 잡아놓고 조치하지 않고 끝내는 것)
    - Inheritance hell : 상속에 대한 지옥!
    - Gold plating (금도금) : 사용자가 요구하지 않은 쓸모없는 기능들을 붙이지 말자
    - Premature optimization : 섣부른 최적화는 악의 근원이다.

- Object-oriented design 5원칙
    - SOILD
        - SRP - Single responsibility principle (가장 중요)
            - 하나의 책임을 가지도록 설계하기
            - 오직 하나의 객체는 하나의 역할을 하도록 설계해라
        - OCP - Open/closed principle
            - 확장에는 열리있고, 수정에는 닫혀있어야한다
        - LSP - Liskov substitution principle
            - 어떠한 객체를 상속받은 sub-class도 부모 class와 같은 동작을 하는게 어울려야한다.
            - 예제) 사람은 동물의 sub-class이다. 동물이 '먹는다'라는 행동을 할 경우, 사람도 '먹는다'라는게 성립한다.
            - 예제) 정사각형은 직사각형의 sub-class가 되면 안된다. 너비와 높이를 각각 바꾸게 되면 정사각형의 성질이 깨지기 때문이다.
        - ISP - Interface segregation principle

            - 인터페이스에는 그 범주안에 들어가는 모든 하위객체들이 공통으로 지닌 특성만을 기술해야한다.
            - 예제) 동물 interface에 fly()라는 메소드를 넣으면 개구리는 fly()할 수 없는데 구현해야한다.
            - 인터페이스를 쪼개어 해결할 수 있다.
        - DIP - Dependency inversion principle
            - 하위 객체가 상위를 보고 구현하는것은 괜찮지만 상위객체가 하위를 보고 구현하는 일은 없도록 하자

- 상세 코딩 이슈
    - 루틴 사용의 주요 이유
        - 복잡성을 줄인다
        - 코드중복을 피한다
        - 세부 구현을 숨긴다
        - 변경 효과를 최소화한다
        - 재사용 가능한 코드를 만든다
    - 루틴 설계시 고려사항
        - 높은 응집도(?) : 내부끼리는 강하게 뭉쳐있어야 한다.
        - 적절한 길이 : 500줄이 넘어가는 루틴은 오류를 발생시킬 확률이 높다
        - 매개변수의 갯수: 5,6개까지는 괜찮지만, 그 이상은 너무 많다.
            - 매개변수 고려사항: 유사한 루틴, 매개변수의 경우 일관된 순서를 유지하라
            - 루틴 매개변수의 수를 7개 정도로 제한하라
            - 모든 매개변수를 사용하라
            - 매개변수로 넘어온 값으로 계속 작업하여 그 것을 return하는 행동은 해서는 안된다
    - 좋은 루틴 이름
        - 하는일을 명확하게!
        - 의미 없고 모호한 동사를 피한다. (너무 광범위하다)
            - HandleCalc, PerformService, ProcessInput
        - 숫자로 구분하지 않는다
            - OutputUser1, OutputUser2
        - 동사 + 객체 이름 형태로 만든다
            - printDocument, CalcMontlyRevenues
        - 반의어를 정확하게 사용하라
            - add/remove, increment/decrement, open/close
    - 변수 초기화
        - 가능하면 선언될 때 초기화하라
        - 가능한 변수가 처음 사용되는 곳과 근접한 곳에서 초기화한다.
        - 가능한 final, const를 적극적으로 사용하라
        - 컴파일러의 경고를 활용한다
    - 변수 사용
        - 변수의 “수명”은 가능한 짧게 유지한다
        - 각 변수를 한 가지 목적을 위해서 사용하라
        - 변수에 숨겨진 의미를 갖지 않게 하라
    - 변수 이름

        - 변수가 표현하고자 하는 것을 정확하게 설명하고 있는가?
        - 적절한 길이
        - 루프에서 i,j,k보다는 의미가 있는 변수를 사용한다
        - 긍정적인 boolean변수 이름을 사용한다.
        - 적절한 축약어 사용
        - 콩글리쉬를 쓰지말자
    - 데이터 사용
        - Magic Number를 피하라
        - 서로 다른 데이터 타입간의 비교하지 않는다.
        - 컴파일러의 경고에 주의를 기울인다,
    - 조건문
        - 상수를 비교문 왼쪽에 놓는다

- 소프트웨어 품질
    - 개발자 테스트
        - 단위 테스트, 컴포넌트 테스트, 통합 테스트
        - TDD
        - 개발자 테스트의 한계
            - “깨끗한 테스트”가 되기 쉽다 (의도한 케이스만을 테스트하기 때문이다)
            - 테스트 커버리지를 낙관적으로 바라본다
            - 정교한 테스트 커버리지를 하지 않는다
        - 테스트 자동화
    - 복잡도 Complexity
        - 코드 신뢰성, 오류 발생 확률과 관련
        - 복잡도를 낮추기 위한 방법
            - 두뇌 훈련으로 처리 능력을 향상
            - 프로그램의 복잡도와 집중해야 하는 대상을 줄인다
        - 복잡도 측정 : decision point를 세는 방법으로!
            - 1로 시작,
            - if, while, for, and, or 에서 +1
            - case문에서 + 1

- Refactoring
    - 나쁜냄새
        - 중복된 코드
        - 긴 메소드
        - 거대한 클래스
        - 긴 파라미터 리스트
        - 확산적 변경 : 특정 부분 하나를 바꾸면 함께 바꿔야하는 체인이 엄청 많은 경우
        - 산탄총 수술 : 변경할때 마다 너무 많은 것을 수정해야 하는 경우
        - 기능에 대한 욕심
        - 데이터 덩어리
        - switch문 : 복잡도를 높이는 행동이다.
        - 추측성 일반화 : 별 의미 없는 추상클레스

        - 메시지 체인 : 다른 클래스를 통해서 사용할 수 있도록 체인이 걸려있는 경우
        - 미들 맨 : 앞에서 얘기한 메세지 체인만을 해주는 클래스, 연결만을 위해 사용하는 의미없는 클래스
        - 부적절한 친밀 : 특정 두 클래스가 너무 tight하게 연결되어있는 경우 (관계를 끊거나 합치거나)
        - 거부된 유산 : 잘못된 상속구조
        - 주석 : 주석이 많이 붙어있다는 것은 코드가 복잡하는 것을 의미한다.
    - 테스트 케이스나 코드가 확실하고 Refactoring을 해도 문제를 일으키지 않을것 같을 때 해야한다.
    - 자주 사용되는 Refactoring 기법
        - Extract method
        - Extract class
        - Inline class
        - Move method
        - Move field
        - Introduce explaining variable : 변수의 이름으로 현재 들어간 값이 무엇인지 중간에 확인하는 방법
        - Remove data value with object : data가 뭉치로 돌아다니면 class로 묶을 수 있다
        - Replace array with object
        - Replace magic number with symbol constant
        - Encapsulate collection : collection이 함께 돌아다닐때는 객체화하는것도 좋다
        - Rename method
        - Introduce parameter object : 파라미터를 오브젝트로 바꾸는것
        - Remove setting method : setter,getter를 항상 만들 필요는 없다
        - Replace constructor with factory method
        - Pull up/down field
        - Pull up/down method
        - Extract subclass/superclass
        - Extract interface
    - 이클립스에서 refactoring을 일부는 제공된다

- Continuous Integration
    - Jenkins를 이용하여 daily build체크를 할 수 있고, code coverage나 정적분석을 할 수 있다.
        - Build Test
        - Code style check
        - Test coverage
        - Static code analysis
        - LOC count
        - Code complexity
        - Duplicate code

- Comment (주석)

    - 스스로 설명하는 코드가 되는게 가장 이상적이다!
    - 코드와 일치하는 Comment
    - 복잡한 로직에 대한 부연 설명
    - Function header는 달아주는게 좋다
    - 일기쓰듯 쓰는 제멋대로 쓰는 주석은 삼가하자

- 참고한 책 목록
    - 실용주의 프로그래머 (앤드류헌트)
    - Code Complete (스티브맥코넬)
    - 켄트백의 구현패턴 (켄트백, 마틴파울러 등)
    - Refactoring (마틴파울러)
    - Design Pattern (GoF)
    - Head First Design Pattern
    - 테스트 주도 개발 (켄트백)
    - Effective Java 2/E (조슈아 블로크)