12 창발성

* 착실하게 따르기만 한다면 우수한 섫계가 나오는 4가지 규칙
  + 모든 테스트를 실행한다
  + 중복을 없앤다
  + 프로그래머의 의도를 표현한다
  + 클래스와 메서드의 수를 줄인다.
* 설계규칙 1 : 모든 테스트를 실행하라
  + 문서로는 시스템을 완벽하게 설계했지만, 시스템이 의도한 대로 돌아가는지 간단한 방법이 없다면, 문서 작성을 위해 투자한 노력에 대한 가치는 인정받기 힘들다.
  + 테스트를 철저히 거쳐 모든 테스트 케이스를 항상 통과하는 시스템은 테스트가 가능한 시스템이다.
  + 테스트가 불가능한 시스템은 검증도 불가능한다. 검증이 불가능한 시스템은 절대 내놓으면 안됨
* 설계규칙2~4 : 리펙터링
  + 테스트케이스를 모두 작성했다면 코드와 클래스를 정리해도 괜찮다. 구체적으로는 코드를 점진적으로 리펙터링 해나간다.
  + 새로 추가하는 코드가 설계 품질을 낮추는가> 그렇다면 깔끔하게 정리 후 테스트 케이스를 돌려 기존 기능을 꺠뜨리지 않았다는 사실을 확인한다.
  + 코드를 정리하면서 시스템이 깨질까 걱정할 필요가 없다. 테스트케이스가 있으니까.
  + 리펙터링 단계에서는 소프트웨어 설계 품질을 높이는 기법이라면 무엇이든 적용해도 괜찮다.
  + 응집도 이고 결합도 낮추고 관심사 분리, 시스템 관심사 모듈로 나누고 함수와 클래스 크기 줄이고 더 은 이름을 선택 하는 등 다양한 기법 동원
* 중복을 없애라
  + 우수한 설계에서 중복은 커다란 적.
  + 중복은 추가 위험, 불필요한 복잡도를 뜻하기 때문 주복은 여러가지 형태로 표출된다. 똑 같은 코드는 당연히 중복이다.
  + 비슷한 코드는 더 비슷하게 고쳐주면 리팩터링이 쉬워짐
* 표현하라
  + 소프트웨어 프로젝트 비용 중 대다수는 장기적인 유지보수에 들어간다. 코드를 변경하면서 버그의 싹을 심지 않으려면 유지보수 개발자가 시스템을 제대로 이해해야 한다. 하지만 시스템이 복잡해지면서 유지보수 개발자기 시스템을 이해하느라 보내는 시간이 늘어남
  + 따라서 코드는 개발자의 의도를 분명히 표현해야 한다. 개발자가 코드를 명백하기 짤수록 다른 사람이 그 코드를 이해하기 쉬워짐
  + 우선 좋은 이름을 선택한다. 이름과 기능 완전히 딴판인 클래스나 함수로 유지보수 담당자를 놀라게해선 안된다.
  + 둘때 함수와 클래스 크기를 가능한 줄인다. 작은 클래스와 작은 함수는 이름 짓기도 쉽고, 구현하기도 쉽고 이해하기도 쉽다.
  + 셋째 표준 명칭을 사용한다. 예를 들어 디자인 패턴은 의사소통과 표현력 강화가 주 목적이다.
  + 넷째 단위 테스트 케이스를 꼼꼼히 작성한다. 테스트 케이스는 소위 예제로 보여주는 문서다. 다시말해 잘 만든 테스트 케이스를 읽어보면 클래스 기능이 한 눈에 들어온다.
  + 표현력을 높이는 가장 중요한 방법은 노력이다. 흔히 코드만 돌린 후 다음 문제로 직행하는 살계가 너무 흔한데, 나중에 읽을 사람을 고려해 조금이라도 읽기 쉽게 만들라는 충분한 고민은 거의 찾기 어렵다.
* 클래스와 메서드 수를 최소로 줄인다.
  + 중복을 제거하고 의도를 표현하고 srp를 준수한다는 기본적인 개념도 극단으로 치닫으면, 득보다 실이 더 많아진다. 클래스와 메서드 크기를 줄이자고 조그만 클래스와 메서드를 수없이 만드는 사례도 없지않다.
  + 때로는 무의미하고 독단적인 정책 탓에 클래스 수와 메서드 수가 늘어나기도 한다. 클래스마다 무조건 인터페이스를 생성하라고 요구하는 구현 표준이 좋은 예다.
  + 목표는 함수와 클래스 크기를 작게 유지하면서 동시에 시스템 크기도 작게 유지하는데 있다. 하지만 이 규칙은 간단한 설계 규칙 네 개 중 우선순위가 가장 낮다. 다시 말해 클래스와 함수 수를 줄이는 작업도 중요하지만, 테스트 케이스를 만들어 중복을 제거하고 의도를 표현하는 작업이 더 중요하다는 것.
* 결론
  + 경험을 대신할 단순한 개발 기법이 있을까? 없다. 하지만 이 장, 아니 이 책에서 소개하는 기법은 저자들이 수십년 동안 쌓은 경험의 정수다. 단순한 설계 규칙을 따른다면 우수한 기법과 원칙을 단번에 활용할 수 있다.