Out of the Tar Pit 요약문

20160169 최수민

Section 1 - Introduction

1. large-scale의 SW system을 개발하고 유지하는 것의 가장 큰 문제는 Complexity이다.
2. Section 2-7에서는 complexity의 종류와 특징에 대해, Section 8-10에서는 이러한 complexity를 해결하기 위한 전략에 대해 소개한다. Section 11-12는 다른 접근 방식과 이를 비교하고 결론을 소개한다.

Section 2 - Complexity

1. SW와 관련된 대다수의 문제의 근원은 Complexity이다. 그 이유는 SW 개발에서의 다양한 문제를 피하기 위해선 시스템을 이해(think and reason about the system)해야 하는데, Complexity가 시스템을 이해하기 어렵게 만들기 때문이다.
2. 이러한 Complexity를 제거하는 것, 즉 Simplicity를 가지는 것은 어렵다.

Section 3 - Approaches to Understanding

1. System을 이해하는 방식에는 크게 두 가지가 있다.
2. Testing
    a. System을 그 바깥에서 이해하려는 시도이다.
    b. 테스트에서는 특정 input set만을 사용하기 때문에, 그 외의 input에 대한 system이나 component의 내부 동작에 대해선 알려주지 않는다는 내재적인 한계가 있다.(specification의 정확도에 의존적이다.)
    c. 또한 테스트를 bug의 존재를 알려줄 수는 있지만, bug가 존재하지 않는다는 것을 알려줄 수 없다.
3. Informal Reasoning
    a. System을 내부에서 이해하려는 시도이다.
    b. 사용될 수 있는 범위가 제한적이고 정확하지 않을 수 있다.
4. 둘 중엔 Informal Reasoning이 훨씬 중요하다. testing을 향상시키면 더 많은 에러를 발견할 수 있다. 하지만 Informing reasoning을 향상시키면 시스템을 더 잘 이해하게 되므로 더 적은 에러가 발생시킬 수 있다.

5. system을 이해하려는 모든 방법은 각각의 한계를 가진다. 이 때문에 simplicity는 더욱 중요하다. Simplicity를 추구하면 이후의 system을 이해하려는 모든 시도를 훨씬 용이하게 만든다.

Section 4 - Cause of Complexity

1. Complexity의 주요 원인은 세 가지가 있다.
2. State
    a. state의 존재 자체가 program을 이해하기 어렵게 만든다.
    b. Testing에 미치는 영향
        i. 특정 state에서 수행되는 test는 다른 state에서의 system이나 component의 동작에 대해 알려주는 게 없다. 특히 internal hidden state가 있는 경우, 테스트는 이와 무관하게 system이 항상 동일한 방식대로 수행된다는 가정에 기댄다.
        ii. Testing의 내재적인 문제(input)와 state로부터 비롯되는 문제는 모두 system에 불확실성을 가져오게 된다.
    c. Informal reasoning에 미치는 영향
        i. Informal reasoning은 일종의 case-by-case mental simulation을 수행하는 것으로 볼 수 있다. 그에 따라 state의 개수가 커질수록, mental simulation은 금세 복잡하고 어려워진다.
        ii. state를 직접적으로 사용하지 않는 procedure가 stateful procedure보다는 훨씬 이해하기 쉽겠지만, 그런 procedure여도 만약 stateful procedure를 내부적으로 호출한다면 결국 state를 고려해야 한다(contaminated).
3. Control
    a. 순서가 중요하지 않은 일들이 있지만, 대부분의 전통적인 프로그래밍 언어는 암시적인 순서가 있다. 그에 따라 시스템이 무엇을 해야 하는지(what)보단 시스템이 어떻게 동작해야 하는가(how)를 구체화해야 한다.
    b. Informal reasoning에 미치는 영향
        i. 코드를 읽을 때 컴파일러처럼 순서를 고려하여 이해한 뒤, 그 이후에 순서가 무시 가능한지 파악해 제거하는 추가 작업을 수행해 복잡해진다.
        ii. 또한 순서가 중요한지 안 중요한지에 대한 결정의 실수로 굉장히 미묘하고 찾기 어려운 버그가 발생할 수도 있다.
    c. Concurrency에 대한 문제(Test, Informal reasoning에 모두 영향을 미침)
        i. Concurrency에 대한 가장 흔한 모델은 shared-state concurrency이다. 이는 explicit synchronization이 제공되는 방식이다.
        ii. Informal reasoning의 경우, 고려해야 하는 시나리오를 추가하는 데 어려움을 겪는다.

iii. Testing의 경우, 시스템이 동일한 상태에서 test를 시작하더라도 결과가 consistent하지 않을 수 있다.

4. Code Volume
    a. 말 그대로 코드의 사이즈를 의미한다. State나 Control에 비해 2차적인 원인이긴 하지만, code volume은 complexity 중 측정하기 쉬운 형태이며, 다른 complexity 원인과 상호작용을 많이 하기 때문에 고려해야 한다.
    b. 두 주요 원인인 state, control을 효율적으로 관리하면 code volume에 따라 complexity가 비선형적으로 증가하지 않을 수 있다.
5. 그 외의 다른 원인
    a. Complexity가 Complexity가 낳는 경우
        i. 시스템을 명확히 이해하지 못해 발생하는 모든 복잡성. e.g. Duplication. 특히 시간 압박이 있을 때.
    b. Simplicity가 어려움
        i. simplicity를 달성하기 위해선 상당한 노력이 요구된다.
    c. Power corrupts
        i. language가 powerful할수록 system을 이해하기 어려워진다. language가 알아서 처리하는 부분이 많아 이로부터 실수나 남용이 발생할 수 있다.

Section 5 - Classical approaches to managing complexity

1. complexity를 관리하기 위한 classical approach는 programming languages의 세 가지 방식으로부터 살펴볼 수 있다.
2. Object-Orientation(Imperative approach)
    a. Von-Neumann 스타일(state-based)의 계산을 수행한다. 즉 OOP의 모든 형태는 state에 의존한다.
    b. State 측면
        i. State 그 자체
            1. Object = some states + set of procedures for accessing and manipulating. 이는 Encapsulation으로 불린다.
            2. 문제
                a. 동일한 state를 조작하는 여러 accessing/manipulating procedures가 있을 경우 제약 조건이 여러 위치에 적용되어야 할 수도 있다.
                b. Encapsulation 기반의 constraint는 객체 1개에 적용되는 데 편향되어 있어 여러 객체와 관련된 복잡한 제약 조건을 적용하기 어렵다.

        ii. State and identity
            1. Object identity - object의 속성과 관계 없이 object는 uniquely identifiable entity로 인식된다(intensional identity).
            2. 문제
                a. object가 stateful abstraction을 제공한다면 합리적이지만, mutability가 필요하지 않은 경우 custom access procedures를 통해 external identity를 도입하게 된다. 이 때 domain-specific equivalence concepts은 standard idea of an equivalent relation(e.g. transivity)를 보장하지 못한다.
                b. object identity는 state의 사용을 기반으로 하고 있어 unavoidable하다. 이는 reasoning을 할 때 external-intensional identity를 교차해 생각해야 해 혼동될 수 있는 complexity를 추가한다.
    c. Control
        i. standard sequential control flow와 shared-state concurrency mechanism을 제공한다. 이로부터 비롯되는 complexity가 그대로 적용된다.
        ii. actor-style language는 동시성의 message-passing model을 사용하기도 하는데, 이는 informal reasoning을 일부 쉽게 만들어주지만 널리 퍼져있지는 않다.
3. Functional Programming
    a. stateless lambda calculus를 기반으로 한다. 이는 Turing machine과 동일한 computational power를 가진다. (본문에선 pure functional programming을 기준으로 설명)
    b. State
        i. functional programming은 state를 피해 referential transparency를 가진다.
        ii. 장점
            1. state로부터 비롯된 문제들(testing, informal reasoning 포함)을 해결한다.
    c. Control
        i. implicit left-to-right sequencing을 가지므로, implicit sequencing에 대한 문제는 그대로 가진다.
        ii. 장점
            1. explicit looping보다는 functionals(e.g. fold, map)을 이용한 control의 추상적 사용을 장려한다.
            2. concurrency에 있어서도 transparency로 인해 보다 안전하다.
    d. Kinds of State

        i. 직접적으로 state를 가지진 않지만, procedure에 extra parameter를 전달함으로써 비슷한 효과를 가질 수 있다.
        ii. 문제
            1. 만약 추가 매개변수가 일종의 collection이라면 mutable variables set을 simulate할 수 있게 된다. referential transparency를 유지함에도 불구하고 reasoning이 어려워질 수 있다.
            2. 하지만 이는 extreme case로, hidden, implicit, mutable states를 갖지 않는 데에는 큰 장점이 있다.
    e. State and Modularity
        i. stateful framework에서는 다른 components를 수정하지 않고도 state를 추가할 수 있다. 반면 functional programming에서는 state를 추가하기 위해선 additional parameter를 추가해야 한다. 이는 complexity(short-cut)와 simplicity 사이의 trade-off 관계이다.
        ii. 이는 functional programming의 주요 강점이자 약점이다. system이 어떤 식으로든 state를 필요로 하는 경우가 있을 수 있다.
4. Logic Programming
    a. problem에 대한 statements만을 서술하는 방식이다. 이상적인 경우는 raw axioms를 받아 solution을 확인하거나 찾는데 사용하는 infrastructure가 있으며, system을 실행하는 것은 각 solution의 formal proof를 만드는 것과 동일하다.
    b. State
        i. Pure logic programming은 mutable state를 사용하지 않지만, 많은 프로그램들은 일부 stateful mechanisms도 제공한다. 이는 referential transparency를 희생하는 것이고, imperative languages와 마찬가지 문제를 겪을 수 있다.
    c. Control
        i. pure Prolog 언어의 경우, sub-goal을 처리하는 left-to-right implicit ordering과 clause application간의 top down implicit ordering이 존재한다. 이는 control flow로 인한 informal reasoning의 어려움을 그대로 가져오게 된다.
        ii. modern languages들은 control에 보다 많은 flexibility를 제공하기도 한다. Oz의 경우 특정 control strategies를 프로그래밍할 수 있는 기능을 제공하기도 한다.

Section 6 - Accidents and Essence

1. Complexity는 크게 두 가지 종류로 나눌 수 있다.

2. Essential Complexity
    a. 유저의 문제 자체에 내재되어 있는 Complexity. 이상적인 환경에서 유저가 원하는 것을 제공하는 SW system을 생산하고 유지하기 위해 SW 개발팀이 고려해야 하는 것이다. 즉 유저가 모르는 영역은 essential일 수 없다.
3. Accidental Complexity
    a. Essential complexity가 아닌 모든 complexity. 현실의 개발에서는 어느 정도 accidental complexity를 포함하게 된다.
4. 이러한 complexity는 SW의 내재된 특성이 아니며, accidental complexity를 최대한 제거해야 한다.