DEVIEW 2020

Swift로 코딩하면 크래시 안난다면서요?

김창기 NAVER

CONTENTS

1. Swift로 바꿨는데 크래시가 줄지 않았어요

동적인 언어인 Objective-C로 작성된 코드의 안정성은 개발자의 역량에 크게 의존한다

1.1 개발자의 문제인가?

DEVIEW 2020

현실적인 문제

- 개발자의 수요 증가가 고급 개발자 양성의 속도를 앞지른다
- 모든 개발 조직에서 고급 개발자만을 투입해서 개발 할 수는 없다
- 고급 개발자라 하더라도 빠르게 코드를 생산 할 수 없다
- 발생한 문제를 빠르게 확인 / 수정하기 어렵다
- 애플의 입장에서는 앱스토어에 불안정한 앱들이 올라오는 것을 방치할 수 없다

1.2 도구를 개선해서 생산성을 올리자

DEVIEW 2020

선택

- 코드베이스를 옮기더라도 생산성을 올릴 필요가 있음
- 개별 개발자의 역량 보다 도구가 안정성을 담보하게 할 필요가 있음
- 정적이고 강한 타입 언어는 코드의 안정성을 도구로 상당 부분 확보할 수 있음
- 익숙치 않은 스타일의 언어를 다수의 개발자가 익숙한 스타일로 바꿀 필요가 있었음

1.3 크래시가 줄지 않았어요

DEVIEW 2020

마이그레이션 후의 현실

- 애플이 홍보한 것 같이 드라마틱한 크래시 감소는 없는 것 같다.
- 크래시의 종류가 바뀌었을 뿐
- 역시 개발자 역량의 문제인가?

1.4 개발자의 문제로 둘 것인가?

DEVIEW 2020

바뀐 상황에서 다시 돌아보기

- 크래시를 개별 건으로 보고 고쳐나가는 것은 한계가 있음
- 발생한 크래시를 모두 조사하여 어떤 패턴을 가지는지 확인해봄
- 동일한 패턴의 크래시가 반복된다면 **가이드라인**을 만들어 방어할 수 있지 않을까?

1.5 어떤 가이드라인을 만들 것인가?

DEVIEW 2020

가이드라인에 대한 가이드라인

- 개수가 **많지 않아야** 한다. 숙지 할 수 있는 분량이어야 한다.
- **단순**해야 한다. 따르는데 혼란이 있어서는 안된다.
- **해결책**을 함께 제시해야 한다.
- 할 수 있으면 **자동화** 해야 한다.
- 맴버들의 **공감**을 얻어야 한다. 자발적인 의지가 있어야 효과적이다.

DEVIEW 2020

2. 어떤 코딩 습관이 우리 앱을 불안정하게 만들었을까요? - 룰을 만들어 봅시다

Swift라고 해도 특정 부분에서
코드의 안정성을 저해하는
언어 자체의 속성이 존재한다.

2.1 Optional

DEVIEW 2020

사라진 것 처럼 보이나 사라지지 않은 null과의 싸움

- Objective-C도 C의 확장이기에 null과의 싸움에서 자유로울 수 없었음
- Optional은 **null이 될 가능성**을 표현함
- Optional을 잘못 처리하면 null 때문에 발생한 크래시와 동일한 크래시를 만듦
- 개발자들이 force unwrap을 너무 쉽게 생각함

2.1.1 Optional은 Optional 답게

DEVIEW 2020

Optional을 force unwap하면 안된다

- 어떤 경우에도 optional은 force unwap(!)을 하지 않아야 함
- guard let / if let을 귀찮아 하지 않아야 함
- Optional일 필요가 없을 때는 optional로 만들지 않아야 함

2.1.1 Optional은 Optional 답게

DEVIEW 2020

```
override func viewDidLoad() {
    super.viewDidLoad()

    let lastName = String(name!.split(separator: " ").last!)   ≡  Thread 1: Fatal error: Unexpectedly found nil while...
    nameLabel?.text = lastName
    print("\(lastName)")
}
```

2.1.2 IUO는 잠재적 크래시

가능성을 남겨놓고 그럴 가능성이 없다는 말을 하면 안된다

- 툴이 outlet들을 기본적으로 IUO로 만들어 준다
  → Optional로 명시적으로 바꾸는 것을 추천
- loading된 UI component들은 unload될 수 있다
  → Unload된 component들을 가리키는 property들은 nil이 된다
- nil이 된 property를 force unwrap 혹은 IUO로 두고 접근하면 크래시

2.1.2 IUO는 잠재적 크래시

DEVIEW 2020

```
@IBOutlet var nameLabel: UILabel!
var name: String! = "Chang-Gi, Kim"

override func viewDidLoad() {
    super.viewDidLoad()

    let lastName = String(name.split(separator: " ").last!)
    nameLabel.text = lastName
}
```

Thread 1: Fatal error: Unexpectedly found nil while implicitly unwrapping an Optional value

2.1.3 캐스팅을 안전하게

DEVIEW 2020

as! 대신 as?를 사용해야 한다

- 설령 100% 확신을 한다고 해도 캐스팅을 할 때 as! 대신 as?를 사용해야 함

```
@IBAction func okButtonTapped(_ sender: Any) {
    let button = sender as! UIButton        Thread 1: signal SIGABRT
    button.isEnabled = false
}
```

Swift 코드에서 “!”를 발견하게 된다면
그곳엔 잠재적인 크래시가 있다고 보면 된다.

2.2 Arithmetic Operation을 조심하자

DEVIEW 2020

많이 사용되지만 run-time에서 방어하기 어려움

- Overflow
- Divide by zero

2.2.1 Overflow

DEVIEW 2020

Swift arithmetic operators don’t allow values to overflow

- Overflow가 발생하면 크래시
- run-time에서 방어 불가
- 연산 후 만들어지는 값의 범위에 각별히 주의할 필요가 있음.
- 필요시 별도의 operator 사용 (&+ 같이 &가 붙어있는 연산자)

2.2.2 Divide by zero

DEVIEW 2020

나눗셈 연산시 제수에 주의를 해야 한다

- Divide by zero 상황에서 코드에서 처리할 기회를 주지 않고 크래시
- 코드에서 0을 매직넘버로 사용해서 나누는 경우는 없을 것임
- 다른 연산을 통해 나온 값을 사용할 때 의심이 되면 반드시 확인이 필요

2.3 32bit 디바이스에서만 발생하는 크래시

DEVIEW 2020

아직 32bit 디바이스들이 사용되고 있다

- 64bit가 대세이지만 32bit 장비도 많이 존재
- Int 같이 디바이스에 따라 32bit/64bit로 가변적인 타입 존재
- Int에 64bit 값을 넣으면 64bit에서는 정상 동작하지만 32bit 디바이스에서 크래시
- 크기가 가변적인 타입은 들어가는 값의 범위를 꼼꼼하게 확인해야 함
- 크기가 64bit가 확실할 때는 Int64 같은 크기가 지정된 타입을 사용해야 함

2.4 Array에 숨어있는 위험

DEVIEW 2020

```
override func viewDidLoad() {
    super.viewDidLoad()

    let lastName = String(name.split(separator: " ")[1])   Thread 1: Fatal error: Index out of range
    nameLabel.text = lastName
}
```

2.4 Array에 숨어있는 위험

DEVIEW 2020

Array의 범위를 벗어나지 않게 주의

- subscript로 array의 element에 접근 할 때 index가 범위를 벗어나지 않게 주의
- 첫 값/마지막 값을 접근 할 때는 subscript보다 first, last 선호
- Extension으로 안전하게 접근할 수 있는 subscript 추가하여 사용

```
subscript(safe index: Index) -> Iterator.Element? {
    guard indices.contains(index) else {
        return nil
    }
    return self[index]
}
```

2.5 Range에 숨어 있는 위험

DEVIEW 2020

Range를 동적으로 만들때 min/max에 주의한다

- max를 min 보다 작은 값으로 지정하면 크래시
- 연산을 통해 min/max값을 생성 할 때 문제가 발생 가능성이 있음

2.6 하부는 여전히 Objective-C

Objective-C 코드에서 exception을 발생시키면 막을 수 없다

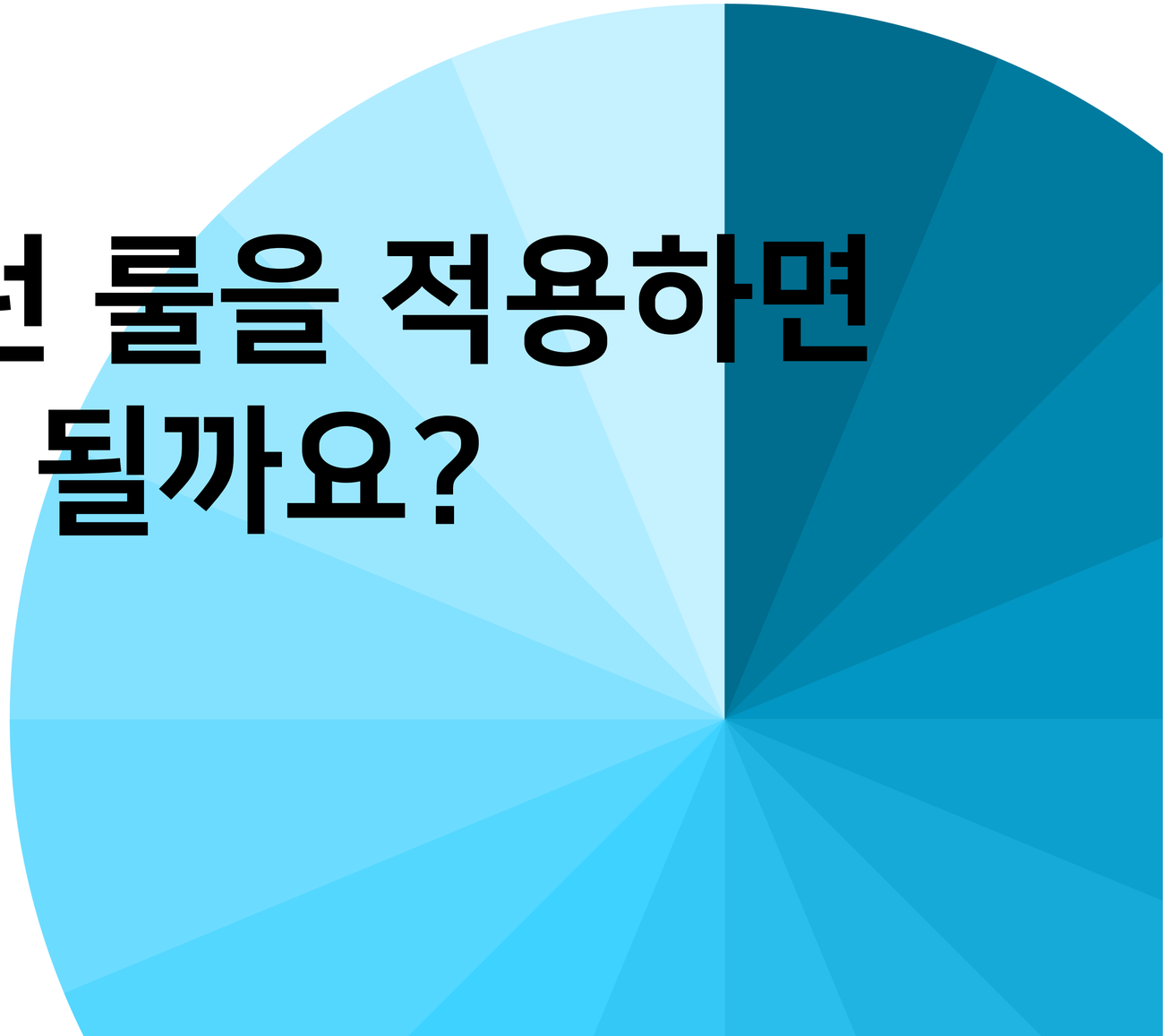
- Swift의 error handling은 Objective-C의 exception과 다르며 호환되지 않음
- UIKit, FoundationKit 등 기존의 framework들은 Objective-C로 되어있고 run-time에 exception을 발생시킴
- Exception의 발생이 우려스러운 상황에서는 exception을 다룰 수 있는 utility 함수의 도움을 받아 처리 할 필요가 있음

2.6 하부는 여전히 Objective-C

DEVIEW 2020

```objc
@implementation ExceptionCatcher

+ (nullable NSException *)tryBlock:(nonnull void (^)(void))block
{
    @try
    {
        block();
    }
    @catch (NSException *exception)
    {
        return exception;
    }

    return nil;
}

+ (nullable id)execute:(nonnull id (^)(void))block
{
    @try
    {
        return block();
    }
    @catch (NSException *exception)
    {
        return exception;
    }

    return nil;
}

@end
```

3. 실제로 이런 룰을 적용하면 어떻게 될까요?



2.1 실행시 마주치는 문제들

DEVIEW 2020

습관은 바꾸기 쉽지 않다

- 명시적 저항 : 어떻게 설득시킬 것인가?
- 습관 : 어떻게 체득해서 습관화 시킬 것인가?
- 기존 코드들에 대한 정리

2.2 SwiftLint를 이용해서 걸러내기

DEVIEW 2020

자동화 할 수 있는 부분은 자동화 하자

```
force_unwrapping
implicitly_unwrapped_optional

force_cast:
  severity: error
force_try:
  severity: error
force_unwrapping:
  severity: error
implicitly_unwrapped_optional:
  severity: error
```

2.3 코드리뷰를 통해 걸러내기

DEVIEW 2020

안타깝게도 모든 것이 자동화되지는 않는다

- SwiftLint로 걸러낼 수 없는 부분들
- 습관이 자리 잡기 전까지는 코드리뷰 시 가장 우선적으로 봐야 할 부분
- 적어도 새로 작성/변경되는 코드에 대해서는 타협하지 않아야 한다

2.4 한번에 다 고칠 수 있을까?

DEVIEW 2020

기존에 작성된 가이드를 따르지 않는 코드가 몇십만 라인이라면?

- 대부분의 경우 별도의 시간이 주어지지 않음 : 고비용
- 경우에 따라 로직을 꽤 많이 건드려야 하는 경우도 발생
- 신규로 작성되는 코드는 100% 따르기
- 변경되는 코드 위주로 우선적으로 수정
- 주기적으로 크래시 로그에서 높은 비중의 크래시를 확인/정리

2.5 효과의 속도

DEVIEW 2020

초반에는 빠르게, 중반 이후는 서서히

- 초반에 큰 비율을 차지하던 부분은 빠르게 제거
- 짧은 시간에 50% 정도의 크래시를 제거 할 수 있음
- 하지만 사용 빈도가 낮은 위치들이 많이 남으면 속도는 감소

4. 남은 숙제들

단순한 문제들이 사라지면
비로소
숨어있던 복잡한 문제들이 보인다.

4.1 단순한 문제들이 사라지면 보이는 것들

DEVIEW 2020

보다 복잡한 문제들

- Objective-C에서 발생하는 exception들을 모두 방어하기는 쉽지 않다
- 기계적으로 크래시를 제거한 부분의 사이드 이펙트로 버그가 나올 수 있다
- Multi-thread에서 생각보다 많은 크래시가 보인다

4.2 Concurrency가 일이키는 문제들

DEVIEW 2020

사용하기 편한 GCD의 배반

- GCD는 사용하기 편함, 그만큼 큰 고민없이 코드에 넣음
- GCD의 상당 부분은 concurrent 이슈를 만들거나 timing 이슈를 만듦
- 자동으로 검출 할 수 있는 툴이 없음
- 그렇다고 사용하지 않을 수도 없음
- 단순한 가이드라인 보다 **정확한 학습**과 **꼼꼼한 코드리뷰**가 필요함

4.2.1 Concurrency문제에 대한 간단 가이드

NDEVIEW 2020

복잡한 문제지만 간단하게 요약하자면…

- Thread로 돌릴 코드는 **pure function**으로 만들면 안전함
  - 그 thread 외부의 변수에 접근하지 않게 한다
  - 잘 제어 할 것이라는 **낙관적 기대를 하면 안됨**
- Thread의 **범위를 최대한 제한**한다
  - thread로 동작하는 코드의 길이를 짧게하고 다른 함수의 call을 줄인다
- 조금 더 복잡해지면 GCD보다는 Operation을 사용하는 것을 고려하자
  - thread가 외부에 영향을 주는 것을 제어하기가 조금 더 편하다
- 특별한 이유가 없다면 main-thread에서 처리하는게 좋을 수 있다

4.3 획일적인 룰로 통제하는 것이 옳을까?

결국 개발자의 문제인가?

- 가이드라인은 **이해** 한 후에 적용해야 **공감**을 얻고 효과적으로 적용 가능
- 결국은 언어 자체의 속성과 그로 인한 코드의 방향성에 대한 **고민**이 필요함
- 일방적인 통제는 바로 효과를 볼 수 있지만 개발자 역량 향상에는 안좋을 수 있음
- 모든 팀원이 보다 안정적인, 보다 생산성을 높일 방법을 **함께** 고민해야 함

Thank You