임베디드 시스템 소프트웨어 개발을 위한 관점지향프로그래밍 방식의 예외원인분석

(Analyzing Exceptions for Embedded System Software Development using Aspect Oriented Programming)

주 재 호 † 김 태 형 ††
(Jae-Ho Ju) (Tae-Hyung Kim)

요 약 소프트웨어에서 예외가 발생할 때 원인분석을 해야 할 때가 있다. 이를 위해서 문제상황을 반복 재현하면서 관찰하게 되는데, 이 때 문제가 발생하는 양상이 규칙적이지 않다면 원인을 분석하는 것이 쉽지 않다. 특히 임베디드 시스템 혹은 이동통신단말기의 경우처럼 개발용 컴퓨터와 개발목적 장치가 분리된 경우에는 원인분석을 위한 시간과 노력이 배가되므로 개발자 부담을 가중시켜서 개발생산성을 극도로 악화시키는 요인이 된다. 본 논문에서는 임베디드 시스템과 이동통신단말기 디버그 중 예외 발생경로를 쉽게 확인할 수 있도록, 관점지향프로그래밍(Aspect Oriented Programming) 방식의 예외원인 분석 방법을 제안한다. 원인이 모호하고 발생빈도가 불규칙적인 예외발생 원인을 규명하기 위해서 예외발생 당시의 함수호출 경로를 추적해야 하므로 소모적인 시간과 노력을 필요로 하는데, 제안하는 방식은 예외발생시 함수호출 경로를 로그 메모리 형태로 즉시 제공해 줌으로써 기존의 디버깅 방법에서 획득할 수 없는 개발자 편의성을 획기적으로 증대시킨다.

키워드 : 관점지향 프로그래밍, 예외원인분석, 예외처리, 임베디드 소프트웨어 개발

Abstract When an unexpected software exception arises, we programmers are to analyze what causes it. Precisely speaking, we need to analyze the cause and property of the unexpected exception. But if exceptions arise irregularly from unknown causes, it is even more difficult for us to handle them, especially in embedded system like mobile phone software development. In this paper, we propose a debugger-friendly analyzing method for exceptions using aspect oriented programming technique. What we need to know upon arising exceptions is the function call history in order to identify the reason for the exceptions. Since programmers used to spend their debugging time on unidentified exceptions, which arise irregularly, our method would greatly improve the embedded software development productivity.

Key words : Aspect Oriented Programming, Exception Analysis, Exception Handling, Embedded Software Development

1. 서 론

관점지향프로그래밍[1]은 소프트웨어 개발 영역에 새로운 패러다임의 모듈화를 제시해 왔는데, 소프트웨어 예외처리 영역에도 관점지향적 모듈화의 관심이 더해지고 있다. 소프트웨어 예외에는 예외적인(exceptional) 것들이 있고 예측하지(unexpected) 못한 것들이 있다[2]. 예측하지 못한 범주의 예외들은 많은 경우에 왜 발생했는지, 그것이 수정되어야 할 문제인지를 판단하기 위한 분석을 필요로 한다. 예컨대, 이동통신단말기 개발과정에서 특별한 에러에 대해 장치를 다시 시작하도록 하는 예외 처리를 했는데, 그 부분으로 인해 장치 수십 대에 한 대 꼴로 부팅 후 바로 재시작하는 현상이 발생한다면 해당 예외의 원인분석 후 디버깅이 필요하다. 그런데 이동통신단말기 소프트웨어 개발자들에게 어떤 범주의 예외들은 원인분석이 어려워 많은 시간과 노력을 투자해야 하는 경우가 있다. 이런 경우를 위해 관점지향프로그래밍을 활용하여 간편하지만 효과적으로 예외원인분석을 하는 방법을 소개한다.

1.1 배 경

개발자들은 예측할 수 없었던 예외들의 근본적인 원인을 파악해서 소프트웨어 결함을 낮추기 위한 노력을 한다. 어떤 경우는 단순히 heap 메모리를 할당 받지 못한 비교적 간단한 문제가 원인이기도 하고, 또 어떤 경우는 무선통신 네트워크 서버와의 동기화 작업에 실패한 비교적 어려운 문제가 원인이기도 하다.

2008년 1분기 기준에서 전 세계 스마트 폰 시장 67%를 점유하고 있는 Symbian OS[3]의 경우, 여러 가지

· 이 논문은 제35회 추계학술대회에서 '임베디드 시스템 소프트웨어 개발을 위한 관점지향프로그래밍(AOP) 방식의 예외원인분석'의 제목으로 발표된 논문을 확장한 것임

† 정 회 원 : 삼성전자 무선사업부 선임연구원
softrock@naver.com
†† 종신회원 : 한양대학교 컴퓨터공학부 교수
tkim@cse.hanyang.ac.kr

논문접수 : 2008년 12월 18일
심사완료 : 2009년 3월 1일

Copyright©2009 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 컴퓨팅의 실제 및 레터 제15권 제5호(2009.5)

상황에 알맞은 예외처리를 위한 패닉(panic) 메커니즘을 가지고 있는데, 이것은 예외가 발생하여 복구할 수 없는 경우 프로그램 실행을 중단시키기 위한 것이다. 이 패닉은 커널 영역, 사용자 영역 등 몇 가지 분류로 나누어지며, 특히 사용자 영역에서는 다시 117개의 항목으로 세분화하고 있다[4]. 이렇게 패닉 범주로 세분화 해놓은 것은 사용자 패닉 처리를 하는 예외 처리 코드 또한 적지 않다는 것을 의미한다.

1.2 연구동기

앞서 예시한 내용과 같이 분석이 필요한 예외들이 임베디드 시스템을 비롯해서 이동통신단말기에서 발생할 때 기존에 흔히 사용해 오던 방법들은 있었다. 그러나 다양한 성질의 예외 분석을 포괄하기 힘들고 또 전문성이 필요하거나 시간이 많이 걸리는 제약사항을 갖고 있었다. 기존의 디버깅 방법들은 다음과 같다.

첫째, 장치에서 로그를 시리얼 포트나 파일로 출력하는 것은 디버그를 위한 특별한 소프트웨어 장치 없이 가장 쉽게 문제에 접근하는 방법이다. 그러나 지나치게 많은 로그는 개발시스템과의 시리얼 통신 또는 파일 출력으로 인해 장치를 느리게 한다. 따라서 원인분석을 위해 로그를 사용할 때는 통상 의심이 가는 부분에 로그를 출력하도록 소스를 수정하고 다시 소프트웨어 소스를 빌드하여 장치에 다운로드 한 후, 다시 예외를 재현하여 출력되는 로그를 확인하는 방법을 반복하면서 문제 원인을 추정하게 된다.

둘째, 장치 에뮬레이터는 개발용 컴퓨터에서 개발목적 장치용 소프트웨어를 개발하고 디버깅하기에 효과적인 환경을 제공한다[5,6]. 이 경우에 예외원인 역시 에뮬레이터상에서 쉽게 분석할 수 있다. 그러나 에뮬레이터에서의 예외 원인 분석의 범위는 제한적이다. 에뮬레이터는 장치를 모방하여 PC에서 동작하는 것이므로 장치 드라이버 및 IrDA, 블루투스, RF(Radio Frequency) 모듈 등의 장치 의존적인 부분의 소스 코드가 실행되지 않거나 혹은 실제 장치의 것과는 다른 모방 코드가 실행되기 때문이다.

셋째, 에뮬레이터가 지원되지 않거나 에뮬레이터에서 문제분석이 어려운 경우, 개발용 PC에 장치를 원격으로 연결하여 실시간 디버깅을 할 수 있다[7]. 그러나 장치 내 디버그 방식은 최초 발견된 예측 불가한 예외를 다시 재현해서 분석하는 방식이므로 예외의 재현성이 좋아야 효과적인 분석이 가능하다. 이것은 원격 디버그 장치를 사용하는 방법에서 취약한 부분이다. 어떤 복잡한 테스트 케이스를 수백 회 반복 테스트를 해야 한 번 나타나는 예외인 경우에는 한번 더 재현하기 위해서 많은 시간을 사용해야 하기 때문이다. 또 예외가 발생했으나 재현하는 방법을 모르는 경우에도 이 방법을 효과적으로 사용하기 어렵다.

마지막으로, 치명적인 상태에 도달한 OS나 응용 프로그램을 재시작 하도록 만드는 크래쉬(crash) 이벤트를 이용하는 방법이다. 이런 치명적인 문제를 해결하기 위해 프로세서 레지스터와 메모리 내용을 분석하는 과정이 그림 1과 같은 크래쉬 덤프이다. 임베디드 시스템에서는 치명적이지 않은 예외라 할지라도 원인분석을 위해서 인위적으로 크래쉬를 발생시켜 덤프를 생성하기도 한다. 그 이유는 예외 발생시 크래쉬 덤프 과정을 거치면 함수 호출 스택을 추적할 수 있기 때문이다, 이 방법은 외부의 간섭 없이 순수하게 장치 그 자체의 실행 중에만 발생하는 예외를 분석을 해야 하는 경우, 예를 들어, ISR(Interrupt Service Routine), 스케줄링 관련 모듈, 서버-클라이언트간 모듈 등, 소프트웨어 시스템 핵심부에서 시간차가 예외발생의 중요 조건인 경우에 효과적이다. 또 최초 예측 불가한 예외 발견 후 다시 재현 시도를 할 필요 없이 바로 원인분석을 할 수 있는 것도 장점이다. 그 분석과정을 요약하면 그림 1과 같다. 먼저 예외가 발생한 지점을 확인하고, 근본적인 원인이 호출자인 상위 함수에 있다면, 함수 호출 경로를 추적해서 예외 원인에 도달할 수 있다. 그러나 프로세서 레지스터 값을 통해 실행 중인 함수의 주소를 알아내고 장치 RAM에 접근하여, 호출 스택을 추적해서 해당 함수의 호출 경로를 다시 구성하는 복잡한 과정을 거쳐야 하는 이 방법은 비교적 많은 시간을 필요로 하고, 시스템 핵심 모듈 디버그에 대한 전문 지식이 없는 개발자들이 사용하기 어려운 단점이 있다.

그림 1 Crash dump를 이용한 예외 원인분석 절차

상기 네 가지 예외원인분석 방법들은 일반적으로 많이 사용하는 방법들이고 유용성을 발휘하지만 특별히 다음의 네 가지 범주에 해당하는 예외들을 특정 디버깅 방법으로 처리하는 데에 어려움과 불편함을 야기시켜 개발생산성을 지극히 저조하게 만든다.

첫째는 재현성이 매우 낮은 예외 발생의 경우이다. 불특정 조건에서 드물게 발생하는 예외는 반복 재현하면서 문제를 관찰하기 어려워 로깅이나 에뮬레이터, 장치 내 디버그 방법을 사용하여 디버깅하는 것이 거의 불가능하다. 인내를 갖고 수백 회 재현 실험을 하거나 운에 맡길 수 밖에 없어서 문제를 해결하기 아주 어려운 경우이다.

둘째는 장치의존성이 매우 큰 예외의 경우이다. RF (Radio Frequency) 소프트웨어 모듈, 무선 LAN, 블루투스, 카메라 등 하드웨어에 밀접하게 관련된 소프트웨어 모듈에서 발생하는 예외는 에뮬레이터에서 원인분석하기가 어렵다.

셋째는 시간차에 민감한 예외의 경우이다. ISR(Interrupt Service Routine), 스케줄링, 서버-클라이언트간 동기화 등은 시간차에 민감한 예외이므로, 실행 중단점을 걸어서 원인을 분석하는 에뮬레이터나 장치 내 디버깅 방식을 사용하기 어렵다.

마지막으로 시스템 핵심 부분에 대한 디버깅이 필요한 예외의 경우이다. 운영체제 커널 디버그가 필요하거나 순수 정상상태의 장치 동작환경에서만 발생하는 문제, 혹은 재현 경로를 알 수 없는 문제를 추적할 때 유용한 방법이지만 전문성이 필요하여 사용할 수 있는 인력이 한정되어 있고 분석에 비교적 긴 시간이 필요한 방법이다.

이런 어려운 점들을 극복하여 예외 원인분석을 빠르고 쉽게 할 수 있는 방법이 있다. 이는 로그 큐(log queue)를 사용하여 예외가 발생할 경우 쉽고 빠르게 함수 호출 경로를 채집할 수 있는 형태로 프로그래밍을 하면 된다. 예외 발생시 함수호출경로에 대한 정보를 개발자가 획득한다면 위에서 열거한 모든 제약사항은 단번에 극복될 수 있기 때문이다. 즉, 개발자는 예외발생시 채집된 함수호출경로를 이용하여, 개발자 숙련도에 관계없이 다양한 계층의 개발자가 예외 발생의 원인을 동일한 방법으로 파악하는 것이 가능하다. 물론 이와 같은 함수호출경로 정보를 저장하는 코드 블록을 개발자가 수작업으로 자신의 소스코드에 삽입하게 하는 것은 새로운 버그를 발생시키고, 프로그램의 복잡도를 증가시키므로, 실효성이 거의 없는 방법이라고 할 수 있다. 본 논문에서는 임베디드 시스템 개발시 특별히 유용한 함수호출경로 채집 방법으로 관점지향 프로그래밍(Aspect Oriented Programming)을 사용한다면, 추가적인 개발비용 상승이나 새로운 버그 발생 가능성 없이 실현 가능하다는 것을 밝힌다.

2. 관련 연구

John Viega와 Jeffrey Voas[8]는 간단하지만 효율적인 AOP 로깅(logging)을 사용하여 보다 신뢰성 있는 소프트웨어를 구축하는 것이 가능할 수 있다는 의미있는 주장을 하였다. 이들은 전통적 예외 처리 코드가 혼잡한 코드를 양산하여 소프트웨어 모듈화를 저해하는 것을 문제로 보았다. 만일, 패키지 내의 모든 함수에서 발생하는 예외를 결합점(join point)으로 잡고 *after throwing* 구문을 효과적으로 사용하는 방식으로 예외처리를 한다면 모듈화가 유지될 수 있다는 주장이다. 하지만 예외처리 코드가 모듈화를 방해하지 않는 방법을 강구하였다고 해서, 본 논문의 서론에서 지적한 예외발생의 원인을 처리할 수 있는 것은 아니다. 우리는 처리되지 못한 예외를 효과적으로 제어하자는 것이 아니라, 그러한 예외가 개발과정에서 발생한 없어져야 하는 버그이므로 개발자가 디버깅한다는 측면에서 AOP 기법을 사용하려는 것이다. 개발자가 원하는 특정 함수의 *before advice* 구문으로서, 예외 발생 시 함수 호출 스택의 내용과 유사한 함수 호출 경로를 보여줄 수 있도록 코드 블록을 삽입할 수 있다.

또한 Lippert와 Lopes[9]는 실험적인 연구를 통해 614개의 클래스를 갖고 약 44,000여 라인에 이르는 특정 프레임워크의 구현내용을 분석한 결과, 실제로 예외 감지 및 처리를 위한 LOC(Lines of Code)가 전체 코드의 무려 10.9%에 해당하는 2,786 LOC라는 것을 분석한 후, 여기에 예외 탐지 및 처리를 위한 모든 부분을 AOP 기법을 이용해서 재구현할 경우 예외 처리에 해당하는 코드 라인 수를 전체의 2.9%로 줄일 수 있음을 보여주었다. 이것은 예외처리를 위해 삽입된 코드가 전체 코드를 혼잡하게 만들어서 모듈화된 설계 내용을 심각하게 훼손하지만, AOP 방법을 이용하여 예외처리를 한다면 이러한 모듈성 훼손을 4배 이상 개선할 수 있음을 의미하는 것이다.

본 저자가 조사해본 바에 따르면, 지금까지 예외처리를 위한 AOP 기법에 관한 연구는 모두 소프트웨어 신뢰성 향상을 위한 예외처리의 불가피성과 예외 처리 관련 코드가 모듈화를 저해하는 모순적인 상황을 회피하기 위한 방법으로 수행되어 왔다. 본 논문에서는 이와 같은 전통적인 목적뿐 아니라, AOP 방법을 이용하면 전통적인 디버깅 방법으로는 실현하기가 어렵지만, 개발자들의 디버깅 능력에 크게 도움이 되는 함수호출경로 채집을 위한 부가 코드를 관심있는 예외 상황에 유효 적절하게 삽입함으로써, 개발 생산성을 증대시킬 수 있음을 보여준다. 여기에 필요한 로그큐(log - queue)를 제안하며, 이를 관점(aspect)으로 효과적으로 모듈화할 수 있다.

3. 로그큐(Log-Queue)의 구성 방법

함수호출경로는 돌발적으로 발생할 예외상황에 대비하여 로깅에 기초하여 구성될 것이다. 일반적인 로깅은 장치를 시리얼 통신 장치로 개발용 컴퓨터에 연결해서 원격으로 로그를 전송하거나, 장치 내부에 파일로 저장한 후 읽어보는 방식이다. 그러나 제안하는 방법에서는 힙 메모리에 하나의 로그큐를 구현한다. 그리고 모든 함수 내지는 관심 있는 모듈의 함수에 log 출력 구문을 삽입하여 큐에 로그가 출력되도록 하는 방식이다. 로그큐는 FIFO 형식이므로 한정된 크기의 큐에서 최신의 함수 호출 경로를 유지하다가 예외 발생 즉시 사용자에게 알리면서 로그큐의 내용을 파일로 출력하여 개발자에게 제공할 수 있다. 이 방법은 로그 내용을 RAM에 출력하게 되므로 시리얼 출력이나 파일 출력보다 속도면에서 유리하다. 또한 최초 예외 발생시 바로 함수호출경로를 읽을 수 있으므로 (1) 예외의 재현성에 크게 관련되지 않고, (2) 목적 장치에서 예외발생시 장치 내에서 즉시 정보를 얻으므로 장치 의존적인 예외가 발생해도 문제없이 원인을 분석할 수 있으며, (3) 장치 내 디버깅 방식처럼 실행중단점(break)을 사용하지 않으므로 시간차에 의한 예외의 호출 경로를 얻기에 어렵지 않다. 이렇게 함으로써 시스템 핵심 부분 디버깅 기술이 뛰어나지 않은 개발자도 예외가 발생하게 된 함수 호출경로를 쉽게 볼 수 있으므로 용이하게 디버깅할 수 있다. 그림 2는 제안내용의 기본적인 동작 원리를 보여준다. 로그는 어드바이스(advice) 코드가 되어 관점(aspect)으로 모듈화 되고 모든 또는 어떤 함수들의 시작부가 교차점(point-cut)이 되어 로그큐에 함수 호출 경로를 남기게 된다. 마침에 예외가 감지될 때 로그큐에 담겨있던 최신의 함수 호출 경로가 장치 내에 파일로 보관되므로 개발자는 예외 발생시 그 파일을 바로 열어서 함수 호출 경로를 얻을 수 있다.

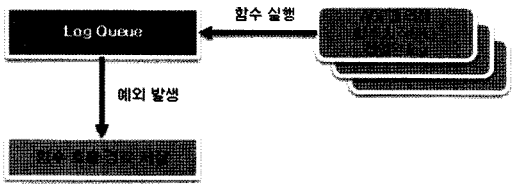

그림 2 로그큐의 동작 다이어그램

4. 관점지향프로그래밍의 적용

제안한 로그큐를 사용하는 방법에서는 함수마다 로그큐로 로깅을 하는 방법이 문제가 되었다. 함수마다 개발자가 로그 구문을 반복해서 넣는 방법은 실질적으로 적용하기 어려운 방법이기 때문이다. 그림 3은 이런 문제점을 관점지향프로그래밍을 통해 해소하는 예이다.

```
aspect FunctionCallLog {
  pointcut all_point () = execution("% %(...)");
    advice all_point() : before() {
   logq.AddCall("+%s:", JoinPoint::signature());
   for (unsigned i = 0; i < JoinPoint::args(); i++)
     logq.AddParam("0x%p, ", tjp->arg(i));
    }
    advice all_point() : after() {
          logq.RmvCall("\n-%s", JoinPoint::signature());
    }
};
```

그림 3 log문을 모듈화하는 aspect 구현

그림 3의 경우에는 모든 함수가 실행되는 부분을 교차점으로 지정하고 *all_point* 교차점의 코드가 실행되기 전과 후에 어드바이스(advice) 코드가 실행되도록 해서 함수 시작시에 로그큐에 로그를 더하고 종료시에 로그큐에서 해당 로그를 빼는 방식으로 함수 호출과정에 대한 기록을 남기도록 했다. 그림 4는 그림 3의 관점 (aspect) 코드가 적용될 기본 프로그램이다. 본 프로그램은 MAX개 만큼 난수를 발생시켜 정렬하는 간단한 프로그램이다.

```
A   void main()
    {
B   [redacted]
      int num[MAX];
      ...
C     try{
      Random(num);
      Sort(num, MAX);
      }
D     catch(char* e) {
      printf("Exception:%s",e);
      logq.Dump();
      }
E     catch(...) {
      printf("unhandled exception\n");
      logq.Dump();
      }
F   [redacted]
      ...
    }

    void Sort(int *nun, int count)
    {
G   [redacted]
      ...
H     Swap(num[i], num[j]);
      ...
I   [redacted]
    }
    void Swap(int *left, int *right)
    {
J   [redacted]
      ...
      if(*left == *right)
K         throw ("Two parameters are
          same in Swap !");
      ...
L   [redacted]
    }
```

그림 4 테스트 프로그램의 기본 소스

본 프로그램에서는 Random함수에서 절대로 중복되는 난수가 생성되지 않도록 해야 하는데 그 부분은 개발자의 실수로 반영되지 않았다. 그러나 개발자는 다행히 Swap함수에서 같은 수를 만나게 될 경우 예외처리 할 수 있도록 K와 같이 throw구문을 사용하였다. C 라인은 main함수에서 전체 프로그램에 해당하는 부분을 try 구문으로 감싸고 있고 예외 발생시 D, E의 두 catch문에서 예외 처리 하도록 하고 있다. 따라서 try문 내부의 어떤 하위 함수에서 예외가 발생하더라도 D, E 두 catch문에 의해 모든 예외가 처리된다. 이는 실제 응용 프로그램 전체에 예외를 catch할 수 있는 덫을 놓았음을 의미한다. 그러나 현 상태로써는 로그큐에 어떤 로그도 들어가지 않는 것처럼 보이지만 그림 4의 관점(aspect) 코드가 각 함수의 시작과 끝 지점에서 로그큐에 로그를 충실히 넣어주고 있다. B, F, G, I, J, L에 해당하는 라인이 각 함수의 시작 및 종료 부분으로 관점(aspect)에 의해 로그가 삽입되는 부분이다. 특별히, Swap함수는 main의 Sort함수를 통해 호출되면서 라인 K와 같이 예외 발생시 throw 구문을 사용하여 예외 처리를 해주고 있다. 라인 K에서 예외발생시 로그큐가 출력한 결과물은 그림 5와 같다. 이와 같은 방법으로, 로그큐 구동으로 인해 시스템이 다소 느려지는 대신 예외 발생시 예외가 어디서 발생했는지, 어떤 함수 실행경로에서 발생 했는지를 커널 개발자부터 응용 프로그램 개발자까지 손쉽게 확인할 수 있게 된다. 소프트웨어 최종 릴리즈 시점에는 로그큐 및 관점코드를 삭제하여 소프트웨어 수행 속도를 확보할 수 있다.

```
Exception : Two parameters are same in Swap!
Dump function call history from Log queue...
==================================
main()
Sort( int* , int  ): 0x3cd72484, 0x3cd72490
Swap(int * , int *): 0x3cd72494, 0x3cd72458
==================================
```

그림 5 로그큐의 출력 예

5. 결 론

앞서 언급한 기존의 예외 원인분석 방법들은 모두가 특정한 여건에서는 이점이 있으나 임베디드시스템과 이동통신단말기의 개발 특성상 몇 가지 부류의 예외에 대해서는 적용이 어렵다는 취약점을 가지고 있었다. 또 장치에서 예외를 즉각적으로 분석하기 위한 크래쉬 디버그 방식은 가장 강력하지만 다양한 계층의 개발자가 공통적으로 사용하기에 부족함이 있었다. 본 논문에서는 기존의 여러 방법으로 나눠진 디버깅 기법을 로그큐 및 관점지향프로그램에 의한 로그 모듈화 방법을 이용하여 단순화함으로써 디버깅을 위한 재현의 어려움 없이 발생한 예외의 원인분석이 즉각적으로 가능한 효과적인 예외 원인분석을 할 수 있음을 보여 주었다. 이와 같은 관점지향방식을 이용한 로그큐 관리를 실제 개발에 적용한다면 대부분의 임베디드 소프트웨어 개발자들이 개발시 가장 고통스러워하는 문제를 확실한 방법으로 해결해 준다고 볼 수 있다. 본 논문에서 제안한 관점지향 프로그래밍 방식을 이용한 예외원인 분석 방법이 실제 개발에 적용될 경우, 개발생산성을 얼마나 획기적으로 향상시킬 수 있는가를 실험적 소프트웨어 공학 원리에 따라 정량적으로 세밀하게 분석해 본다면 매우 흥미로운 결과가 나올 것으로 기대된다. 본 저자의 실제 개발 경험상, 실제 개발자들이 예외원인 분석에 소모하는 시간이 상당한 비중을 차지하고 있는데, 본 논문의 방법을 사용한다면 이러한 불필요한 원인분석 시간은 모두 제거될 수 있기 때문이다.

참 고 문 헌

[1] Kiczales, G., et al.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland. 1997.

[2] HENNESSY, J. Program optimization and exception handling. In Conference Record of the ACM Symposium on Principles of Programming Languages (Williamsburg, Virginia, Jan.). 200-206, 1981.

[3] Symbian Fast Facts Q1 2008. DOI= http://www.symbian.com/about/fastfacts/fastfacts.html

[4] Symbian OS 9.2 System Panic Reference User Category. DOI= http://developer.uiq.com/devlib/uiq_31/sdkdocumentation/doc_source/doc_source/reference/SystemPanics/UserPanics.html#Panics%2euser

[5] Alex Feinman, device debugging and Emulation in Visual Studio 2005. DOI= http://msdn.microsoft.com/en-us/library/aa454884.aspx

[6] Rechard Harrison & Mark Sharkman. Symbian OS c++ for mobile phones: 3. John Wiley and Sons Inc., 306, 2007.

[7] Jane Sales. Developing software for Symbian OS - An Introduction to Creating Smartphone Applications in C++. John Wiley and Sons Inc., 20, 2005.

[8] J. Viega and J. Voas, Can Aspect-Oriented Programming Lead to More Reliable software?," IEEE software 17, pp. 18-21, 2000.

[9] Lippert, M. and Lopes, C.V. A study on exception detection and handling using aspect-oriented programming. In Proceedings of the 22nd International Conference on software Engineering, June 2000.