DOI: 10.3745/KIPSTD.2009.16-D.1.83

# AOP를 이용한 재공학에서의 핫 스팟 탐색과 응용

이 의 성[†] · 최 은 만[††]


## 요 약

현실 세계의 복잡한 비즈니스 로직들이 프로그램 내부에 투영되면서 시스템의 복잡도는 갈수록 높아지고 있다. 이러한 높은 복잡도를 가지는 프로그램도 그 생명주기 동안 재공학을 거쳐야 하는 것은 필연적일 것이다. 다양한 목적으로 가해지는 재공학 작업에서 그 작업의 대상이 되는 핫 스팟 예측은 매우 중요하다. 일반적으로 레거시 시스템의 재공학 작업은 UML과 코드 분석을 기반으로 예측한다. 또한 그 예측 단위는 클래스 혹은 유닛(함수) 단위가 된다. 그러나 함수 내부의 코드의 량이 갈수록 커져가고 있고 더 미세한 핫 스팟을 찾기 위하여 클래스 단위의 탐색보다 더 미세한 부분의 탐색이 필요하다. 본 논문에서는 AOP를 이용한 문장 단위의 핫 스팟 검출 기법을 제안한다. 기존의 기법에서 요구하던 핫 스팟 검출을 위한 UML과 코드 분석, 또한 이 둘 사이의 일치성과 관계없이 동적으로 AOP를 이용하여 레거시 시스템의 실행 정보를 기록하는 동적 이벤트 로그 데이터를 생성한다. 이를 바탕으로 핫 스팟을 예측하고 슬라이싱하는 방법을 제안하였다.


**키워드 : 재공학, AOP, 핫 스팟 탐색**

# Method and Application of Searching Hot Spot For Reengineering Software Using AOP

Ei Sung Lee[†] · Eun Man Choi[††]


## ABSTRACT

Complicated business logic makes program complexity more complicated. It's inevitable that the program must undergo reengineering processes all the way of in its lifetime. Hot spot analysis that has diverse purposes is getting an important question more and more. As a rule, reengineering process is done by UML model-based approach to analyze the legacy system. The smallest fragment of targets to be analysed is unit, that is function or class. Today's software development is to deal with huge change of software product and huge class including heavy quantity of LOC(Lines Of Code). However, analysis of unit is not precise approach process for reliable reengineering consequence. In this paper, we propose very precise hot spot analysis approach using Aspect-Oriented Programming languages, such as AspectJ. Typically the consistency between UML and source is needed code to redefine the modified library or framework boundaries. But reengineering approach using AOP doesn't need to analyze UML and source code. This approach makes dynamic event log data that contains detailed program interaction information. This dynamic event log data makes it possible to analyze hot spot.


**Keywords : Reengineering, AOP, Hot Spot Analysis**

## 1. 서 론

오류 수정, 새로운 요구사항 적용, 알고리즘의 변화 등 다양한 목적으로 재공학 작업이 이뤄진다. 기업 입장에서의 재공학 목적은 소비자의 요구를 발 빠르게 충족시키기 위한 작업이 되기도 할 것이다. 이 때 기업들은 자사가 가지고 있는 현재의 자원을 최대한 유지하며 재공학을 통해 품질 향상을 꾀한다. 이러한 목적은 정확한 핫 스팟[22] 예측에서 시작될 것이다. 핫 스팟이란 시스템에서 아주 중요한 역할을 하는 부분으로 이런 핫 스팟을 수정하려 할 때 원시 코드의 슬라이스 혹은 그 수정의 영향을 받는 원시 코드의 범위를 찾아야 한다.

거대한 시스템의 규모만큼 그 산출물(UML, 소스 코드)도 방대해지면서 재공학을 위한 핫 스팟[19] 검출은 어려워지고 있다. UML과 코드 분석이 선행되어야 하는 기존의 핫 스팟 검출 기법[2]은 레거시 시스템의 개발 초기부터 참여한 개발자가 아니라면 핫 스팟 검출에 있어 어려움을 겪게 된다. 선행 작업에서 시스템에 대한 정확한 이해도를 가져야 성공적인 핫 스팟 검출 작업을 할 수 있기 때문이다. 물론 이런 선행 작업에 있어서 UML과 소스 코드 사이의 일치성은 매우 중요하지만 촉박한 프로그램 개발 환경 속에서 문서화 작업이 높은 신뢰성은 갖기에는 많은 허점들이 있다. 또한

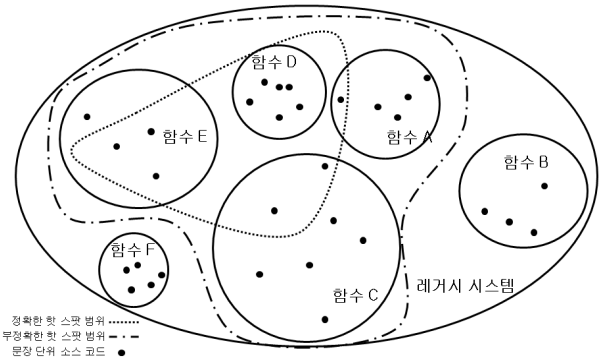
† 정 회 원 : 동국대학교 컴퓨터공학전공 석사과정
†† 정 회 원 : 동국대학교 IT학부 컴퓨터공학전공 교수
논문접수 : 2008년 9월 12일
수 정 일 : 1차 2008년 11월 25일
심사완료 : 2008년 11월 25일

핫 스팟의 검출 단위는 기존의 방법[12~14]에서는 유닛(함수) 단위이기 때문에 불필요한 부분을 담고 있다. (그림 1)은 핫 스팟 검출에서 불필요한 부분들이 발생하는 예를 보여주고 있다. 일반적인 방법의 핫 스팟 검출 단위가 유닛(함수)이기 때문에 불필요한 소스 코드 묶음도 핫 스팟 범위에 속하게 된다. 예를 들어 함수 C의 일부분이 핫 스팟에 해당하지만 유닛 단위의 핫 스팟 검출로 인해 함수 C 전체가 핫 스팟의 범위로 정해지게 된다.

이러한 일반적으로 사용되는 핫 스팟 검출 기법의 단점들을 극복하기 위하여 오류 검출을 위한 동적 슬라이싱 기법[20, 21]이 소개되기도 하였다. 프로그램의 실행 정보를 동적으로 기록하고 기록한 데이터를 분석하여 오류의 범위를 검출하는 방법이다. UML은 프로그램의 구조를 보여주는 것이 주요한 목적이기 때문에 실제 프로그램이 실행되었을 때 그 실행 순서를 UML 형식의 그래프로 보여주는 것은 한계가 있다. 특히 레거시 프로그램에 오류가 존재한다면 컨트롤 플로우 그래프를 이용한 핫 스팟 예측도 정확성을 잃게 된다. 이렇게 오류 검출을 위한 일반적인 슬라이싱 방법에서 UML을 기반으로 하는 기법들이 프로그램의 실행 정보를 표현하기 힘들기 때문에 동적 슬라이싱 알고리즘[21]을 도입하여 일반적인 핫 스팟 검출 기법의 단점을 보완하려는 시도가 있었다.

이러한 기존 방법의 단점을 보완하기 위해 AOP(Aspect Oriented Program)를 이용한 핫 스팟 검출 기법을 제안한다. 기존의 방법과 달리 UML과 코드 분석 없이 동적으로 프로그램을 추적하여 프로그램 속에 산재하여 있는 핫 스팟을 검출한다. 레거시 시스템의 개발 초기부터 참여하지 않은 개발자는 기존 방법을 이용하여 UML을 분석하고 핫 스팟을 찾기는 결코 쉬운 일이 아니다. 또한 분석을 위한 많은 경험도 필요할 것이다. 기존 방법보다 빠르고 명확한 핫 스팟 범위를 예측하는 이 방법은 UML과 소스 코드 사이의 일치성을 요구하지 않으며 단지 프로그램 각 기능의 실행만으로 시스템 내부의 상호작용 정보를 동적으로 기록한다. 이렇게 기록한 자료를 동적 이벤트 로그 데이터라고 하며 이 데이터를 바탕으로 동적 슬라이싱 작업을 한다[4]. 결국 동적 슬라이싱 작업으로 정확한 핫 스팟 검출이 가능하다.

2장에서는 일반적인 재공학 기법에 대해 소개하고 단점들을 언급한다. 2장의 단점을 극복하기 위한 AOP를 이용한 재공학 기법을 3장에서 소개하고 4장에서 동적 이벤트 로그 데이터를 생성하여 핫 스팟을 검출하는 기법을 자세히 다루었다.

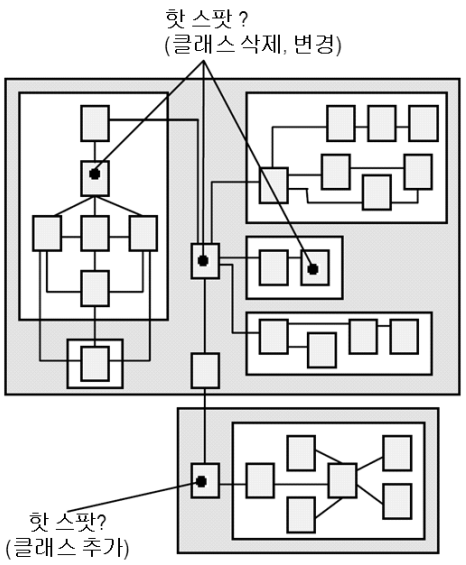



(그림 1) 부정확한 핫 스팟 검출 결과의 예

## 2. 일반적인 재공학 기법

변경의 요인이 발생하였을 때 제일 먼저 필요한 작업은 UML을 철저히 검증하여 변환의 범위를 분석[12]하는 것이다. 특히 시스템이 복잡한 구조를 가지며 시스템 구조의 중추가 UML로 표현되어 있다면 재공학 작업은 UML에 많은 부분을 의존하게 된다. (그림 2)에서 보이는 것과 같이 UML을 통해 어느 부분에 변환이 가해질지 예상할 수 있다. 그리고 그 변환의 목적은 <표 1>과 같이 다양하다

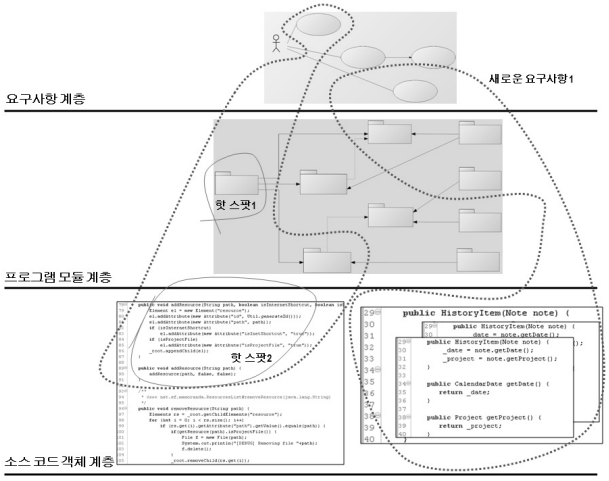
재공학을 위한 핫 스팟 검출에 있어 중요한 점은 그 정확성에 있을 것이다. 정확한 위치와 정확한 범위 측정이 재공학 작업의 성공에 크게 작용할 것이다. 이러한 정확도에 있어서 기존의 방법은 한계점을 갖고 있다. 핫 스팟을 검출해야 하는 경우는 (그림 3)에서와 같이 요구사항의 변경에서




(그림 2) UML 기반의 핫 스팟 예측

〈표 1〉 추상 수준에서의 주요 변환 요소

| 변환 대상 | 변환 종류 | 변환 대상 | 변환 종류 |
|---|---|---|---|
| 변수 | 형 변환<br>변수영향범위 변환<br>추가, 삭제 | 조건문 | 실행 조건 변환<br>루프 범위 변환 |
| 함수 | 반환값 변환<br>구현방법 변환<br>파라미터 변환<br>접근자 변환<br>함수영향범위변환<br>추가, 삭제 | 상속구조 | 상속 단계 변환<br>상속 범위 변환 |
| 클래스 | 상속구조 변환<br>추가, 삭제 | | |




(그림 3) 요구 사항, 프로그램 모듈, 소스 코드 사이의 추상적 연관 관계

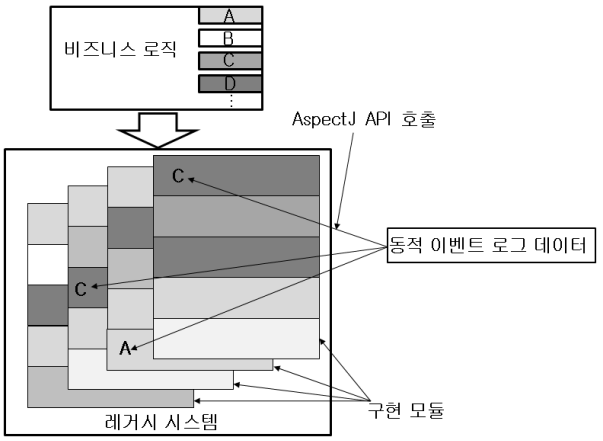
소스 코드 객체 계층의 수정까지 다양하다.

일반적인 방법으로 새로운 요구사항이나 프로그램 모듈 수준의 핫 스팟을 UML을 통하여 검출이 가능하다. 하지만 (그림 3)의 핫 스팟2와 같이 핫 스팟의 범위가 특정한 모듈 전체가 아닌 하나의 모듈에 속에 있는 소스 코드 일부일 경우 그 측정은 어려워진다. 또한 실행 과정에 따라 객체들이 새롭게 생성되고 사용되는 객체지향 언어에서 UML을 기반으로 프로그램 내부의 상호작용을 파악하고 함수 파라미터와 같은 함수 내부에서 사용되는 변수들을 파악하기는 매우 어렵다. 그리고 하나의 모듈의 부피가 커질수록 기존 방법의 핫 스팟 검출 단위에는 많은 불필요한 부분이 끼어들 가능성이 커지게 된다는 것도 기존 기법의 단점이 될 것이다. 즉, 클래스 다이어그램 자체를 이해하여 프로그램 내부에서 일어나는 객체들 사이의 상호작용을 상세하게 분석해 내기는 매우 어려운 작업일 것이다.

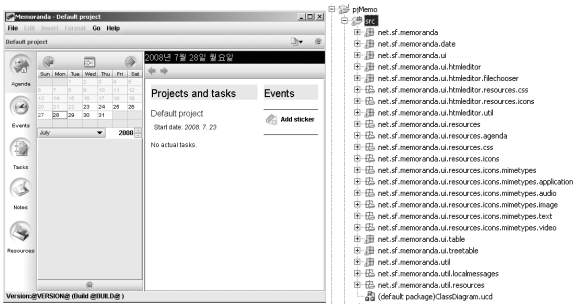
기존의 논문들에서는 UML을 기반으로 정적 프로그램 행위분석[15]을 하거나 UML 기반의 레거시 시스템 수정[16]을 하는 기법들을 소개하고 있는데 이러한 방법들 모두 UML을 철저히 분석해야 하는데 많은 노력이 필요하다. 또한 그 기능상의 한계점도 안고 있다.

## 3. 재공학을 위한 AOP 적용

AspectJ는 레거시 시스템 추적을 위한 많은 API를 제공한다. 동적으로 시스템 외부에서 레거시 시스템을 추적할 수 있는 API의 응용을 통하여 각 기능마다 요구하는 프로그램 내부의 함수, 클래스의 호출 시기, 호출 정보, 코드 상의 위치 등을 동적으로 검출하고 기록할 수 있다. 그리고 이러한 기능들이 코드 인스트루먼트 기법을 이용하는 것과 달리 원시 코드가 그대로 유지되면서 가능하게 된다. 핫 스팟 검출을 위한 소스 코드 수정 없이 AOP가 웨이빙 되면서 시스템 분석이 가능하기 때문에 핫 스팟 검출을 위한 원시 코드 수정 작업과 검출 작업 후의 소스 코드 내부의 검출을




(그림 4) AspectJ를 이용한 동적 로그 데이터 생성



(그림 5) 재공학 대상 시스템의 메인 화면과 프로그램 상의 패키지 구조

위한 코드 삭제 작업이 사라지게 된다.

(그림 4)는 AOP를 핫 스팟 검출 기법에 어떻게 적용시킬지를 간략하게 보여주고 있다. 시스템 전반에 걸쳐 녹아들어 있는 비즈니스 로직을 AspectJ API를 이용하여 추적의 대상이 되는 모듈들을 동적으로 추적하여 그 모듈들의 다양한 상호작용 정보를 로그 데이터로 남긴다. 이러한 기법을 적용할 레거시 시스템은 (그림 5)와 같다. 3장에서 수행하는 실험의 대상이 되는 (그림 5)의 시스템은 개발자의 스케줄을 전산화하여 관리할 수 있는 프로그램의 메인화면이다.

(그림 5)의 시스템을 대상으로 본 논문에서 제시하는 AOP를 이용한 핫 스팟 검출 기법을 적용할 것이다. 개발자 개개인의 개발 일정을 관리할 수 있는 이 시스템[5]은 164개의 클래스 파일과 23개의 패키지로 구성되어있다. 이 시스템에 재공학을 위한 핫 스팟 예측 작업이 필요하다고 가정하고 기존 기법의 한계점과 AOP를 이용한 핫 스팟 검출 기법의 장점들을 기술한다.

(그림 6)에서 이러한 AOP를 이용한 핫 스팟 검출 예를 간단하게 보여주고 있다. 레거시 시스템의 특정한 대화상자 창을 활성화 하였을 경우 AspectJ API는 어떤 함수가 호출되는지 그 함수의 위치, 구현 정보 등을 실시간으로 동적 이벤트 로그 데이터에 기록한다. 이렇게 기록된 로그 데이터를 바탕으로 실행한 특정 기능이 변환의 대상이 된다면 그 핫 스팟은 소스 코드 상의 어떤 위치인지 상세히 분석해


(그림 6) 특정 기능 실행에 따른 동적 이벤트 로그 데이터 생성과 동적 슬라이싱

낼 수 있다.

재공학 작업은 시스템의 구조와 기능 분석으로 시작하여 코드 변환과 테스트로 마무리된다. 이러한 작업에서 코드 분석을 위한 시간을 AOP를 응용한 도구를 이용하여 줄일 수 있다면 재공학에 큰 도움이 될 것이다.

### 4.1 핫 스팟 검출을 위한 동적 이벤트 로그 데이터

AspectJ를 이용하여 동적으로 테스트 범위를 예측하는 기법[7~11]은 여러 논문들에서 소개된 바 있다. 기존의 소개된 기법들은 데이터 플로우 그래프를 이용한 테스트, AspectJ로 검출 가능한 오류의 종류 등과 같이 테스트를 중심으로 연구되었다. 즉 예측이 아닌 AOP를 이용한 더 효과적인 오류 검출과 테스트와 관련된 연구들이다.

본 논문에서는 이러한 기술들을 바탕으로 AspectJ의 강력한 API를 이용하여 프로그램의 핫 스팟 예측을 UML이 제공하지 못 하는 상세한 부분까지 검출한다. 기존의 논문에서 소개된 방법을 확장하여 테스트와 오류검출 그리고 변환을 위한 핫 스팟 예측을 AOP를 이용하여 구현한다. 이러한 기법을 이번 장에서 소개한다.

(그림 7)의 AspectJ 소스 코드는 UML의 도움 없이 핫 스팟 검출을 가능하게 한다. 핫 스팟 검출의 목적을 언어변환(UI 언어변환)에 두었다고 가정하였다. 언어변환 작업은 전 시스템에 걸쳐 UI를 생성하는 객체를 모두 수정해야 하기 때문에 시스템 전반에 걸쳐 언어변환과 관련된 함수가 사용되고 있어서 언어변환을 핫 스팟 검출의 목적으로 정하였다. 또한 실험의 대상이 되는 이 시스템에서 UI 객체 선언을 위해 Java 언어 패키지의 내부 API와 함께 사용자가 정의한 함수도 모두 사용하고 있어 다양한 추적 결과를 얻을 수 있다. (그림 7)의 코드에서 14번째 줄에 pointcut을 getString 함수와 setText 함수로 한정한 것을 확인할 수 있다. 이 두 함수는 UI 객체를 생성하며 객체의 문자열을 정의(Local.getString("History forward"))하는 함수이다. 핫 스팟 검출의 목적이 언어변환이기 때문에 추적 대상 함수를 이 두 함수로 한정하였다. 16번째 줄과 23번째 줄 사이의

```
8  public aspect OpticLogger {
9      int seq = 0;
10     OpticLogger (){
11         System.out.println("aspect constructor");
12     }
13     pointcut traceMethods()
14         : (call(* *.getString(..)) || call(* *.setText(..))) && !within(OpticLogger);
15
16     before() : traceMethods(){
17             Signature sig = thisJoinPointStaticPart.getSignature();
18             archiveLog("seq " + seq);
19             archiveLog("thisJoinPointINFO : "+createParameterMessage(thisJoinPoint));
20             archiveLog("SourceLocation : "+ thisJoinPointStaticPart.getSourceLocation());
21             archiveLog("sig.getName() : "+sig.getName());
22             archiveLog("sig.getDeclaringType().getName() : "+sig.getDeclaringType().getName());
23             archiveLog("----------------------------------------");++seq;
24     }
25     private String createParameterMessage(JoinPoint joinPoint){
26         StringBuffer paramBuffer = new StringBuffer("\n\t[This : ");
27         paramBuffer.append(joinPoint.getThis());
28         Object[] arguments = joinPoint.getArgs();
29         paramBuffer.append("]\n\t[Args: (");
30         for(int length = arguments.length, i=0; i<length; ++i){
31             Object argument = arguments[i];
32             paramBuffer.append(argument);
33             if(i != (length-1)){
34                 paramBuffer.append(',');}
35         paramBuffer.append(")]");
36         return paramBuffer.toString();
37     }
38     private void archiveLog(String str){
39         try{
40         FileWriter fos = new FileWriter("test.txt",true);
41         fos.write(str+"\n");
42         fos.close();
43         }
44         catch(IOException e){
45             System.out.println("filewirter error : "+e);
46         }
47     }
```

(그림 7) 레거시 시스템의 핫 스팟 검출을 위한 AspectJ 소스 코드

```
3627seq 450
3628thisJoinPointINFO :
3629    [This : net.sf.memoranda.ui.ExitConfirmationDialog[dialog0,C
3630    [Args: (Exit)]
3631SourceLocation : ExitConfirmationDialog.java:60
3632sig.getName() : setText
3633sig.getDeclaringType().getName() : javax.swing.JLabel
3634----------------------------------------
3635seq 451
3636thisJoinPointINFO :
3637    [This : net.sf.memoranda.ui.ExitConfirmationDialog[dialog0,C
3638    [Args: (This action will cause Memoranda to exit)]
3639SourceLocation : ExitConfirmationDialog.java:67
3640sig.getName() : getString
3641sig.getDeclaringType().getName() : net.sf.memoranda.util.Local
3642----------------------------------------
3643seq 452
3644thisJoinPointINFO :
3645    [This : net.sf.memoranda.ui.ExitConfirmationDialog[dialog0,C
3646    [Args: (Do you want to continue?)]
3647SourceLocation : ExitConfirmationDialog.java:68
3648sig.getName() : getString
3649sig.getDeclaringType().getName() : net.sf.memoranda.util.Local
3650----------------------------------------
3651seq 453
3652thisJoinPointINFO :
3653    [This : net.sf.memoranda.ui.ExitConfirmationDialog[dialog0,C
3654    [Args: (<HTML>This action will cause Memoranda to exit<p>Do
3655SourceLocation : ExitConfirmationDialog.java:67
3656sig.getName() : setText
3657sig.getDeclaringType().getName() : javax.swing.JLabel
3658----------------------------------------
3659seq 454
3660thisJoinPointINFO :
3661    [This : net.sf.memoranda.ui.ExitConfirmationDialog[dialog0,C
3662    [Args: (do not ask again)]
3663SourceLocation : ExitConfirmationDialog.java:70
3664sig.getName() : getString
3665sig.getDeclaringType().getName() : net.sf.memoranda.util.Local
3666----------------------------------------
```

(그림 8) AspectJ 코드의 실행 결과를 기록한 동적 이벤트 로그 데이터

코드는 추적 대상 함수의 어떤 정보를 검출할지 API를 이용하여 정의하고 있다. 함수의 실행 횟수, 실행 위치와 파라미터 정보, 소스 코드 라인 상의 실행 위치, 함수의 이름, 구현 위치, 순으로 정보를 검출하고 있다.

(그림 7)의 AspectJ 소스 코드는 레거시 시스템 내부에 녹아 있는 UI 객체 생성 API를 추적하고 있고 그 실행 결과는 (그림 8)과 같다. 총 459번의 함수 호출을 기록한 이 동적 이벤트 로그 데이터는 다양한 추적 결과를 내포하고 있다.

모든 로그 데이터의 일부분을 보여주고 있는 (그림 8)에서는 한 번의 함수 실행에 대해 총 5개의 정보를 보여주고

있다. 시스템 구조상의 함수 위치, 함수의 파라미터 정보, 소스 코드 라인 수준의 호출 위치 정보, 함수 이름, 함수의 구현 위치를 모두 나타내고 있다.

(그림 8)의 451번째 호출정보에서 다양한 정보를 나타내고 있다. 추적된 함수의 위치가 어떤 패키지 안의 어떤 클래스에 위치해 있는지 나타내고 그 다음으로 함수의 파라미터 정보를 나타내고 있다. UI 객체의 문자열을 정의하는 함수를 추적하였기 때문에 문자열을 나타내는 파마리터인 "(This action will cause Memoranda to exit)"를 나타내고 있다. 다음으로 sourceLocation 기능을 이용하여 추적하는 함수가 파일을 몇 번째 줄에서 호출되었는지 정확히 나타낸다. 그리고 함수의 이름, 구현위치가 나타나고 있다. 451번째 호출 정보에서의 함수의 구현 위치는 nef.sf.memoralda.util 패키지 안의 Local 클래스 파일에서 구현되었다. 이와 달리 453번째 호출 결과에서는 함수의 구현 위치가 javax.swing.JLabel로 나타나 있는데 그 이유는 453번째 추적 함수는 setText이기 때문이다. setText 함수는 Java의 기본 API에서 제공해주는 UI 객체 선언 함수이기 때문에 java의 swing 패키지 안에 구현되어 있는 것으로 나타나고 451번째 호출 정보에서는 getString 함수를 추적 하였는데 이 함수는 사용자가 정의한 함수이기 때문에 그 구현 위치가 다른 것을 확인 할 수 있다. 이 모든 추적 정보는 UML과 소스 코드 분석 없이 이루어졌고 기존 방법에서는 할 수 없던 정보들까지 모두 검출되었다.

### 4.2 동적 이벤트 로그 데이터를 이용한 슬라이싱

4.1장에서 설명한 AspectJ를 이용한 핫 스팟 검출 결과 중 파라미터 추적 결과와 AOP를 이용한 핫 스팟 검출의 효율성을 좀 더 자세히 설명한다. 일반적인 방법을 통하여 프로그램을 정적으로 분석하거나 UML을 통해서 파라미터의 값을 확인할 수 있었다. 하지만 그 파라미터의 개수가 매우 많아지게 되면 코드 인스트루먼트를 위한 방법으로는 파라미터 검출을 위한 코드가 원시 코드에 너무 많이 삽입되기 때문에 파라미터 검출 작업 후의 검출을 위한 코드 삭제 작업의 부담이 발생하게 된다.

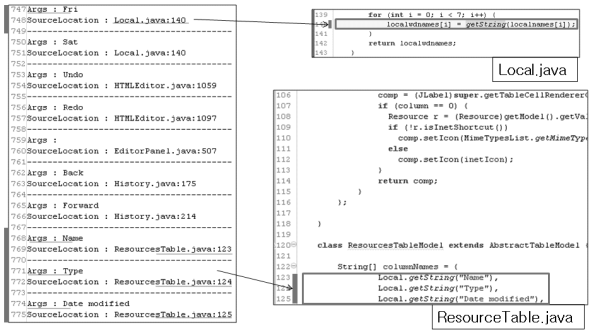
이런 경우 AspectJ를 이용한 파라미터 추적 기법은 프로그램 실행과 함께 실시간으로 함수가 사용하고 있는 파라미터의 값을 확인할 수 있고 그 값의 오류 여부를 파악하여 핫 스팟을 검출 할 수 있을 것이다.

4.1장에서 언급한 언어변환을 목적으로 핫 스팟을 검출한다고 하였을 때 그 파라미터의 값이 매우 중요하다. 레거시 시스템에 언어변환을 가한다고 할 때 가장 중요한 것은 UI 객체들의 문자열 값이 의도한 대로 언어변환(외국어로 변환)이 되었느냐가 관건이기 때문이다. 정확한 언어변환은 그 시스템의 품질을 좌우하게 된다. 프로그램 내부의 작은 오류가 아닌 사용자가 접하게 되는 UI 상의 오류는 그 조건에 따라 큰 피해를 가져올 수 있기 때문이다.

레거시 시스템에 언어변환 작업이 가해지기 전에 (그림 9)와 같은 번역작업이 이루어지게 된다. UI 객체의 문자열

| | A | B | C | D |
|---|---|---|---|---|
| 151 | | HTS | 티커창 | Màn hình ticker |
| 152 | | HTS | 팝업메뉴 설정 | Cài đặt pop up menu |
| 153 | | HTS | 폰트 | Font |
| 154 | | HTS | 프린터 | Máy in |
| 155 | | HTS | 하 | Dưới |
| 156 | | HTS | 화면비율 | Tỷ lệ màn hình |
| 157 | | HTS | 화면설정 | Cài đặt màn hình |
| 158 | | HTS | 화면열기 | Mở màn hình |
| 159 | | HTS | 화면장금 | Khóa màn hình |
| 160 | | HTS | 화면전환 | Chuyển màn hình |
| 161 | | HTS | ○○ 중에서 | Trong 00 |
| 162 | | HTS | ○○차 호가 | Gía chào thứ 00 |
| 163 | | | | |
| 164 | | 공통 | 담당자 | Nhân viên nghiệp vụ |
| 165 | | 공통 | 무관 | Không có liên quan |
| 166 | | 공통 | 본사 | Công ty chính |
| 167 | | 공통 | 본사지점구분 | Phân loại chi nhánh/chính |
| 168 | | 공통 | 작업권한 | Quyền tác nghiệp |
| 169 | | 공통 | 직무종료일 | Ngày kết thúc nghiệp vụ |
| 170 | | 공통 | 직책 | Chức trách |
| 171 | | 공통 | 직책 코드 | Mã chức trách |
| 172 | | 공통 | 직책명 | Tên chức trách |

(그림 9) 프로그램 상의 언어변환을 위한 단어 별 외국에 변환 자료(베트남 언어)



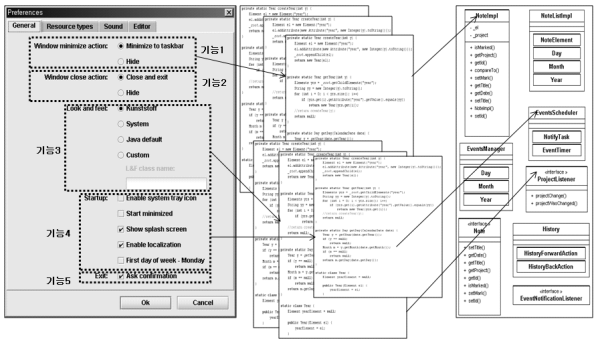

(그림 10) 파라미터 검출과 소스 코드 수준의 핫 스팟 위치 검출을 위한 동적 로그 데이터

에 들어갈 이와 같은 문자열은 수백 개에 이르게 되고 번역작업이 끝난 후 번역된 문자들은 시스템의 언어변환 작업에 사용되게 된다. 이러한 문자열 변환 작업이 시스템 어디에서 일어날 것인지 AOP를 이용하여 그 핫 스팟을 예측하여 본다면 보다 명확한 언어변환 작업 범위를 정의할 수 있을 것이다.

(그림 7)의 AspectJ 소스 코드를 응용하여 언어변환 작업이 가해질 소스 코드 만을 추적하여 (그림 10)과 같은 동적 로그 데이터를 생성하였다. 그리고 이 데이터를 바탕으로 소스 코드를 슬라이싱 하였다. UI의 언어변환을 가할 파라미터와 그 UI가 선언된 소스 코드 상의 위치를 기록한 이 로그 데이터는 UML 분석으로 얻지 못 하는 파라미터와 소스 코드 수준의 핫 스팟 정보를 제공한다. (그림 10)의 왼편에 나타난 동적 로그 데이터에서 768번째 줄과 775번째 줄 사이의 결과에서는 getString 함수의 호출 위치와 각각의 다른 파라미터 값을 나타내고 있다. 이 데이터를 바탕으로 (그림 10)의 오른편에 나타나 있는 것과 같이 소스 코드를 슬라이싱 할 수 있다. 이러한 함수 단위의 슬라이싱이 아닌 소스 코드 라인 단위의 슬라이싱과 호출 할 때마다 다른 파

〈표 2〉 초기 메뉴의 언어변환을 위한 LOC 예측

| 시스템 전체 LOC | 초기 언어변환 LOC | 전체 변환 코드 |
|---|---|---|
| 약 29913 LOC | 약 459 LOC | 약 1.53 % |




(그림 11) 기능 분류에 따른 핫 스팟 검출

라미터를 갖는 함수의 추적은 AOP를 이용한 핫 스팟 검출 기법으로 가능하게 된다. 또한 이러한 예측 결과를 바탕으로 <표 2>와 같이 언어변환을 위한 API가 호출된 라인수를 계산하여 전체 시스템 LOC(Lines of Code) 중 몇 퍼센트 이상이 언어변환으로 인해 그 수정의 대상이 되는지 예측할 수 있다. 물론 이러한 예측에 드는 시간은 기존의 UML과 코드 분석을 통한 시간보다 적게 걸릴 것이다.

이러한 AOP를 이용한 핫 스팟 검출 기법은 재공학을 위한 소스 코드 수정 작업 이전에 시스템의 어느 부분에서 얼마나 많은 수정이 가해질지를 예측할 수 있다. 이러한 예측 기법이 일반적인 기법보다 선호되려면 예측 결과에서 차이를 보여야할 것이다. UML과 코드 분석 없이 핫 스팟 검출이 가능하다는 것은 핫 스팟을 검출하는 과정에도 차이가 있고 결과에도 차이가 존재한다. 결과의 차이를 보이기 위하여 (그림 11)과 같이 크게 5개의 기능을 분류하고 각 기능 수행을 위한 코드를 핫 스팟으로 정의하는 실험하고 비교하였다. 5개의 기능은 프로그램의 기본 환경설정에 해당하는 preference 메뉴의 general 탭에 속해 있는 총 5개의 환경설정 기능이다.

(그림 11)의 각 기능들을 직접 실행하여 이벤트 로그 데이터를 생성하고 그 데이터를 바탕으로 코드를 분석하는 방식으로 AOP를 이용한 핫 스팟 검출 기법을 실험하였다. 같은 기능을 기존의 방법을 이용하여 유닛 단위로 핫 스팟 범위를 검출하였다. 이렇게 검출된 핫 스팟에 해당하는 소스 코드의 라인수를 비교하여 일반적인 유닛 단위의 슬라이싱과 AOP를 이용한 슬라이싱 결과 사이에 차이를 <표 3>에 나타내었다.

<표 3>에서는 AOP를 적용하였을 때의 핫 스팟 크기와 일반적인 방법으로 핫 스팟을 검출하였을 때 그 크기의 차이가 나타나 있다. (그림 11)에서 기능 3을 실행 하였을 때 몇 개의 함수가 호출이 되었고 그 총 호출 횟수 그리고 총 사용된 코드의 수(Lines Of Code)가 <표 3>에 나타나 있다. <표 3>의 좌측에 나타나 있는 AOP를 적용한 핫 스팟 검출 결과는 모두 기능을 직접 실행하여 얻은 이벤트 로그 데이터를 바탕으로 작성되었다. 함수의 호출 횟수나 사용한 문장 단위의 코드는 (그림 7)의 AspectJ 소스 코드를 이용하여 검출하였다. <표 3>의 오른편에 나타나 있는 일반적인 방법의 핫 스팟 분석 결과는 UML과 소스코드를 분석하여 얻은 결과이다. 동적 다이어그램을 통하여 함수의 개수, 호출 횟수를 파악할 수 있을 것이다. 그리고 사용된 소스 코드의 라인 수는 유닛 단위로 검출되기 때문에 그 검출 크기가 문장 단위의 검출을 하는 AOP를 이용한 검출 보다 더 많은 라인수를 나타내고 있다.

<표 3>에 나타난 결과 중 가장 큰 차이는 핫 스팟의 크기를 나타내는 각 기능에 사용된 소스 코드 상의LOC일 것이다. 각 기능에 필요한 함수의 개수나 호출 횟수는 UML을 통해서 분석이 가능하다. 하지만 사용된 LOC의 경우 AOP를 이용하며 문장 단위의 검출이 가능하기 때문에 불필요한 핫 스팟이 삭제된다.

핫 스팟의 검출 목적이 테스트에 있다면 핫 스팟의 범위에서 불필요한 코드들은 불필요한 테스트 작업을 낳게 된다. 핫 스팟의 검출 목적이 코드 변환이라면 이 또한 불필요한 코드 변환 작업을 낳을 것이다. 테스트와 변환 작업의 규모가 커지면서 테스트 작업의 중복 분석[23]도 중요한 연구 주제가 되고 있다. 이러한 테스트 작업의 불필요한 요소를 삭제하기 위하여 핫 스팟의 정확한 검출이 더욱 중요할 것이다.

### 4.3 시스템 기능 단위의 핫 스팟 범위 측정

4.2절에서는 레거시 시스템 내부의 특정한 API들을 한정하여 핫 스팟의 범위를 측정하였다. (그림 10)에 나타난 결과는 추적하려는 API가 이미 정의된 상태에서 핫 스팟의 범위

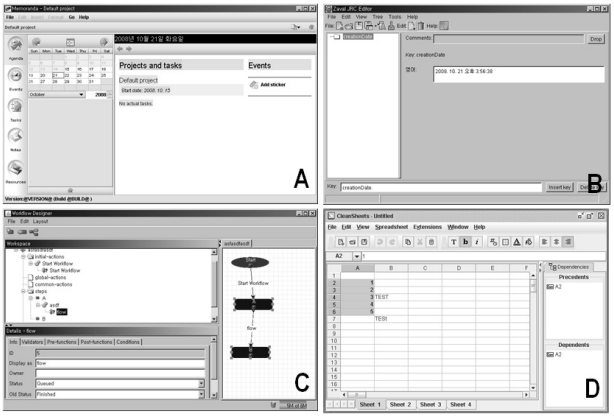
〈표 3〉 AOP를 적용한 핫 스팟 크기 비교

| | AOP 적용 핫 스팟 분석 | | | 비 AOP 적용 핫 스팟 분석 | | |
|---|---|---|---|---|---|---|
| | 함수 개수 | 호출 횟수 | 사용된 LOC | 함수 개수 | 호출 횟수 | 사용된 LOC |
| 기능 1 | 2 | 2 | 2 | 2 | 2 | 10 |
| 기능 2 | 2 | 2 | 2 | 2 | 2 | 10 |
| 기능 3 | 5 | 16 | 16 | 5 | 16 | 60 |
| 기능 4 | 3 | 6 | 6 | 3 | 6 | 18 |
| 기능 5 | 1 | 2 | 2 | 1 | 2 | 6 |

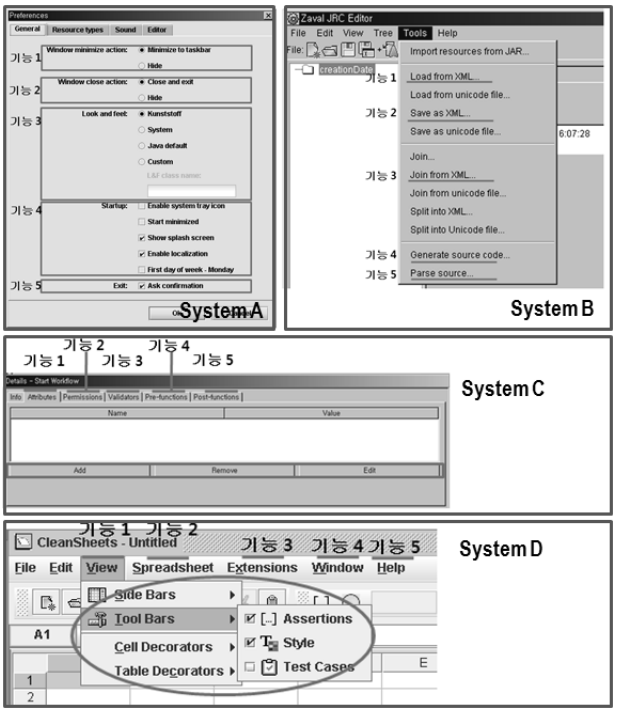
를 측정하는 경우이다. 핫 스팟의 범위를 측정할 때 4.2절의 실험과 같이 항상 사용되는 API가 미리 파악되는 것은 아니다. 레거시 시스템의 어떠한 API가 사용될지 알 수 없는 상황에서 시스템의 기능에 따른 핫 스팟을 측정해야 하는 경우도 발생할 수 있을 것이다. 이러한 경우, 시스템의 API가 아닌 기능을 대상으로 핫 스팟을 검출해야 한다. 하나의 실험 결과로 제안한 방법의 성능을 보이기 어려운 면이 있어 실험 대상을 늘렸으며 그 결과는 4.2절의 <표 3>과 같다.

(그림 12)는 기능 단위의 핫 스팟 검출을 위한 실험에 사용된 4개의 레거시 시스템이다. 4개의 시스템 모두 소스 코드 전체가 공개되어 있고 자바로 구현되었기 때문에 이클립스를 통해 소스 코드를 분석하고 AspectJ를 적용한 실험을 실행하였다. (그림 12)의 4개의 프로그램을 선택한 이유는 프로그램의 분류에 따라 그 구현 방법이 다르기 때문이다. 예를 들어 병원에서 사용되는 프로그램은 그 개발사에 따라 메뉴와 기능은 조금씩 다르지만 수행하여야 하는 핵심기능은 크게 다르지 않기 때문에 설계상의 큰 차이점을 갖기는 힘들다. 다양한 구현 방법으로 만들어진 시스템 분석을 통하여 다양한 실험 결과를 얻기 위하여 각기 다른 목적을 가지는 4개의 시스템을 실험 대상으로 정하였다. (그림 12)의 시스템 A는 3장에 소개하였고 4.2장에서 실험한 시스템이다. (그림 12)의 시스템 B는 자바 소스 코드를 편집하구 컴파일 할 수 있는 개발환경을 제공한다. (그림 12)의 시스템 C는 다양한 분야에서 워크 플로우를 정의하고 그래프로 나타낼 수 있는 기능을 제공한다. (그림 12)의 시스템 D는 엑셀과 같은 스프레드 쉬트를 제공하는 프로그램이다. 각 프로그램의 이름과 크기는 <표 4>를 통하여 나타내었다.

<표 4>에 나타낸 4개의 프로그램에서 각 기능을 분류하여 분류된 기능을 실행하고 생성된 동적 이벤트 로그 데이터를 분석하여 핫 스팟 범위를 측정하였다. 레거시 시스템마다 분류된 기능은 (그림 13)을 통하여 나타내었다. 시스템 A는 프로그램의 환경 설정을 위한 옵션을 5개로 나누어 각 기능이 실행되었을 때 실시간으로 그 실행 정보를 동적 이벤트 로그 데이터에 담아 5개의 각 옵션 설정에 필요한 소스 코드의 범위를 측정하였다. 시스템 B는 작성한 자바 소스 코드의 구조를 XML로 저장하고 로드하는 기능을 시스템 A와 같은 방법으로 측정하였다. 시스템 C는 워크 플로우의 단계와 사전 조건들을 정의하는 기능을 5개로 나누어




(그림 12) 핫 스팟 검출을 위한 실험 대상 레거시 시스템




(그림 13) 기능 단위 핫 스팟 검출을 위한 각 레거시 시스템의 기능 분류

측정하였다. 시스템 D는 프로그램의 메인 메뉴를 5개로 나누어 각 메인 메뉴에 속하는 하위 메뉴들 까지 포함하여 5

〈표 4〉 실험 대상 레거시 시스템의 복잡도 분석

| | 어트리뷰트 개수 | 클래스 개수 | 함수 개수 | 총 Lines OF Code | 맥케이브 사이클로매틱 복잡도 (mean/maximum) |
|---|---|---|---|---|---|
| 시스템 A memoranda | 1,272 | 164 | 1,073 | 21,828 | 2.412/42 |
| 시스템 B osworkflow | 599 | 90 | 1,698 | 22,643 | 3.881/171 |
| 시스템 C jrc-editor | 695 | 382 | 2,442 | 32,206 | 2.324/48 |
| 시스템 D CleanSheets | 323 | 245 | 1,346 | 14,180 | 2.16/105 |

〈표 5〉 기능 단위 핫 스팟 범위에 따른 소스 코드 크기(LOC) 분석 결과

| | LOC 분류 | 기능 1 | 기능 2 | 기능 3 | 기능 4 | 기능 5 |
|---|---|---|---|---|---|---|
| System B osworkflow | 문장 단위LOC | 456 | 154 | 175 | 254 | 196 |
| | 함수 단위 LOC | 592 | 195 | 215 | 274 | 226 |
| | LOC 분류 | 기능 1 | 기능 2 | 기능 3 | 기능 4 | 기능 5 |
| System C jrc-editor | 문장 단위LOC | 174 | 351 | 110 | 327 | 178 |
| | 함수 단위 LOC | 214 | 400 | 164 | 418 | 254 |
| | LOC 분류 | 기능 1 | 기능 2 | 기능 3 | 기능 4 | 기능 5 |
| System D CleanSheets | 문장 단위LOC | 201 | 111 | 200 | 294 | 278 |
| | 함수 단위 LOC | 302 | 223 | 403 | 354 | 387 |

개의 기능으로 정의하고 실험하였다. 각 실험 결과는 UML을 기반으로 하는 일반적인 핫 스팟 측정 방법의 핫 스팟 범위와 AOP 기법을 적용하여 동적 이벤트 로그 데이터를 바탕으로 한 결과를 비교하여 나타내었다. 그 실험 결과는 <표 5>와 같다. <표 5>에 나타난 것과 같이 모든 시스템과 모든 기능의 핫 스팟 검출에서 AspectJ를 이용한 문장 단위 핫 스팟의 크기(LOC)가 더 적은 것으로 나타났다. <표 5>에서 시스템 A가 없는 이유는 4.2절에서 실험하였고 <표 3>을 통해 그 실험 결과를 이미 나타내었다. 실험 분석 결과 시스템 D에서는 함수 단위 핫 스팟 검출 범위가 문장 단위 핫 스팟 검출 범위 보다 35.05%의 불필요한 부분이 포함된 것으로 분석되었다. 나머지 3개의 시스템에서 모두 핫 스팟의 범위에서 차이를 보이고 있다.

AOP를 이용한 핫 스팟 검출 기법이 일반적인 UML을 이용한 핫 스팟 검출 기법보다 더 정확하고 불필요한 코드가 제거되는 이유는 그 검출 단위가 문장 단위이기 때문이며 동적인 추적 기능을 이용하기 때문이다.

(그림 14)의 소스 코드는 시스템 D의 SrcGenerator 클래스의 일부분이다. (그림 13)에 나타난 오버로딩을 이용한 코드는 함수의 파라미터에 따라 실행되는 함수가 변하게 된다. 워크프로우를 정의하고 자동화한 시스템 D에서 makeVarName 함수가 하는 일은 각 워크플로우의 단계를 정의하는 것이다. 이 함수는 실행되는 기능에 따라 동적으로 오버로딩 되어 실행 순서가 변하게 되는데 이 때 AOP는 프로그램의 실행 과정을 동적으로 감지하여 오버로딩 되는 함수들도 어떤 함수가 실행되게 되는지 감지할 수 있다. 하지만 UML을 기반으로 하는 핫 스팟 검출 기법은 클래스 다이어그램 상에 시스템의 구조가 정확히 표현되지 못 하거나 시스템을 잘 이해하지 못 한 테스터가 핫 스팟을 측정하게 된다면 핫 스팟 검출에 있어서 오류가 발생하게 된다.

(그림 14)에 나타난 오버로딩의 경우와 그 함수의 소스 코드 크기는 작지만 이렇게 동적으로 변하는 실행과정의 경우가 많은 객체지향에서 각 기능에 따른 정확한 핫 스팟 검출은 매우 중요하다. 기능에 따른 핫 스팟 검출은 작은 오차들이 모여 큰 차이를 가져올 수 있기 때문이다.

<표 5>는 이러한 작은 오류들이 얼마나 큰 차이를 가져오는지 나타내고 있다. (그림 14)에 나타난 오버로딩을 포함

```
private String makeVarName(BundleItem bi)
{
    String ask = bi.getTranslation("__var");
    if(ask!=null) return ask;
    ask = makeVarName(bi.getId());
    bi.setTranslation("__var", ask);
    return ask;
}

private String makeFunName(BundleItem bi)
{
    String ask = bi.getTranslation("__varF");
    if(ask!=null) return ask;
    ask = capitalize(makeVarName(bi.getId()));
    bi.setTranslation("__varF", ask);
    return ask;
}

private String makeVarName(String key)
{
    String s = key.toLowerCase();
    int j1 = s.lastIndexOf('.');
    if(j1<0) return s;
    int j2 = s.lastIndexOf('.', j1-1);
    if(j2<0) j2 = -1;
    s = s.substring(j2+1,j1) + capitalize(s.substring(j1+1));
    return s;
}
```

overloading

(그림 14) AOP를 이용한 동적 시스템 분석의 예

한 동적으로 변하는 실행과정은 UML을 기반으로 하는 핫 스팟 검출 기법이 객체지향 시스템 분석에서는 오류를 범할 수 있다는 것을 의미한다. 그리고 AOP를 적용한 핫 스팟 범위 분석 기법은 그 오류의 폭을 줄여줄 것이다.

## 5. 결론 및 향후 연구

UML과 코드 분석을 기반으로 하는 재공학 기법[12~16]은 여러 논문을 통해 소개되어왔다. 기존의 이러한 방법의 단점을 극복하기 위한 정적 분석 방법[4]도 소개된 바 있다. 이런 논문들에서 제시한 방법과는 달리 본 논문에서는 프로그램의 실행 정보를 동적으로 추적하고 기록하였다. 또한 기존 기법에서 나타낼 수 없었던 프로그램 소스 코드 수준의 추적이나 파라미터 검출도 나타내었다. 그리고 소스 코드와 UML의 분석 없이 핫 스팟을 예측하고 슬라이싱 할 수 있다는 것이 본 논문에서 제시하는 방법의 큰 장점이라고 할 수 있을 것이다. 이러한 장점을 실험으로 증명하기 위하여 4장을 통하여 실험 결과를 나타내었다. 4.2절의 <표 2>와 4.3절의 <표 5>를 통하여 가장 중요한 실험 결과인

핫 스팟의 크기(LOC) 감소를 확인할 수 있었다.

재공학을 위한 핫 스팟 검출 기법에 AOP를 적용한 본 논문은 핫 스팟의 범위를 측정하는데 중점을 두었다. 앞으로 이 기초 연구를 바탕으로 수많은 핫 스팟 중 우선순위를 두어 레거시 시스템에 어느 부분을 우선적으로 변환해야 하는지에 대한 연구를 진행할 계획이다. 변환을 가할 때 핫 스팟이 아닌 다른 모듈에 오류가 발생하지 않도록 핫 스팟과 비핫 스팟 사이의 관계를 파악하는 작업도 이번 논문의 결과와 같이 자동화 하는 기법을 적용할 수 있을 것이다.

## 참 고 문 헌


[1] Isabel Michiels, "Using AOP Techniques for Dealing with Legacy Systems," *PROG*, Vrije Universiteit Brussel (VUB), Belgium.

[2] L.C. Briand, Y.Labiche, L.O'Sullivan, "Impact Analysis and Change Management of UML Models," *Proceedings of International Conference on Software Maintenance*, pp. 256-265, 2003.

[3] "AspectJ - an aspect-oriented extention to the Java programming language" http://www.eclipse.org/aspecj/

[4] Xiangyu Zhang, "Dynamic Slicing Long Running Programs through Execution Fast Forwarding," SIGSoft 06/FSE-14, November 5-11,2006.

[5] "Memoranda - open source cross-platform diary manager and a tool for scheduling personal projects," http://memoranda.sourceforge.net/

[6] M. M. Lehman and L.Belady. *Program Evolution - Processes of Software Change,* London Academic Press, 1985.

[7] Ot´avio Augusto Lazzarini Lemos, Cristina Videira Lopes, "Testing Aspect-Oriented Programming Pointcut Descriptors," *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, pp.33-38, 2006.

[8] Jianjun Zhao, "Data-Flow-Based Unit Testing of Aspect-Oriented Programs," *Proceedings of the 27th Annual International Computer Software and Applications Conference*, pp.188-197, 2003.

[9] J. S. Bekken, Roger T. Alexander. "A Candidate Fault Model for AspectJ Pointcuts," *Proceedings o 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pp.169-178, 2006.

[10] Lionel, C. Briand, Yvan Labiche, Johanne Leduc, "Tracing Distributed Systems Executions Using AspectJ," *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp.81-90, 2005.

[11] John Stamey, Jr.Bryan Saunders, "Unit Testing and Debugging with Aspects," T*he Journal of Computing Sciences in Colleges,* pp.47-55, 2005.

[12] L. C. Briand, Y. Labiche, L. O'Sullivan, "Impact Analysis and Change Management of UML Models," *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pp.256-265, Sept., 2003.

[13] F. Fioravanti, G. Migliarese, P. Nesi, "Reengineering Analysis of Object-Oriented System via Duplication Analysis," *Proceedings of the 23rd International Conference on Software Engineering*, pp.577-586, 2001.

[14] Hassine, J.; Rilling, J.; Hewitt, J.; Dssouli, R., "Change Impact Analysis for Requirement Evolution using Use Case Maps," *Principles of Software Evolution Eighth International Workshop*, pp.81-90, 2005.

[15] Kollmann, R.; Gogolla, M, "Capturing dynamic program behaviour with UML collaboration diagrams," *Fifth European Conference on Software Maintenance and Reengineering*, pp.58-67, 2001.

[16] Ye Wu, Offutt, J, "Maintaining evolving component-based software with UML," *Proceedings of Seventh European Conference on Software Maintenance and Reengineering*, pp.133-142, 2003.

[17] Dave Thomas, Andy Hunt, "Mock Objects," *IEEE Software*, Volume 19, Issue 3, May-June, pp.22-24, 2002.

[18] Gerard Meszaros, *xUNIT TEST PATTERNS :REFACTORING TEST CODE*, Addison-Wesley, 2007.

[19] James R. Cody, Kevin A. Schneider, Thomas R. Dean, Andrew J. Malton, "HSML: design directed source code hot spots," *Proceedings of 9th International Workshop on Program Comprehension*, pp.145-154, 2001.

[20] Xiangyu Zhang, Gupta R, Youtao Zhang, "Precise dynamic slicing algorithms", *Proceedings of 25th International Conference on Software Engineering*, pp.319-329, 2003.

[21] Xiangyu Zhang, Haifeng He, Neelam Gupta, Rajiv Gupta, "Experimental evaluation of using dynamic slices for fault location," Proceedings of the sixth International Symposium *on Automated Analysis-driven Debugging*, pp.33-42, 2005.

[22] Cordy J.R, Schneider K.A, Dean T.R, Malton A.J, "HSML Design Directed source code Hot Spots," IWPC 2001. *Proceedings of 9th International Workshop on Program Comprehension*, pp.145-154, 2001.

[23] Tao Xie, Jianjun Zhao, Darko Marinov, David Notkin, "Detecting Redundant Unit for AspecJ Programs," *17th International Symposium on Software Reliability Engineering,* pp.179-190, Nov., 2006.

이 의 성

e-mail : palevalentine@dongguk.edu
2007년 동국대학교 컴퓨터공학과(학사)
2009년 동국대학교 대학원 컴퓨터공학과(석사)
2009년~현 재 KT 경영지원팀 사원
관심분야 : AOP, 소프트웨어 테스트, 재공학

최 은 만

e-mail : emchoi@dgu.ac.kr
1982년 동국대학교 전산학과(학사)
1985년 한국과학기술원 전산학과 (공학석사)
1993년 일리노이 공대 전산학과(공학박사)
1985년~1988년 한국표준연구소 연구원
1988년~1989년 데이콤 주임연구원
2002년 카네기멜론대학 소프트웨어공학 과정 연수
2000년, 2007년 콜로라도 주립대 전산학과 방문교수
1993년~현 재 동국대학교 IT학부 컴퓨터공학전공 교수
관심분야 : 객체지향 설계, 소프트웨어 테스트, 프로세스와 메트릭, Program Comprehension