솔루션간 차이에 기반한 적응형 소프트웨어 솔루션의 분리

유찬우 O1, 정우성 1, 이병정 2, 김희천 3, 우치수 1

1 서울대학교 컴퓨터공학부

2 서울시립대학교 컴퓨터과학부

3 한국방송통신대학교 컴퓨터학과

chlorinde@gmail.com, wsjung@selab.snu.ac.kr, bjlee@venus.uos.ac.kr, hckim@knou.ac.kr, wuchisu@selab.snu.ac.kr

Separation of Adaptive Software Solutions Using Variations

Chanwoo Yoo O1, Woosung Jung1, Byungjeong Lee2, Heechern Kim3, Chisu Wu1

1School of Computer Science & Engineering, Seoul National University

2School of Computer Science, University of Seoul

3Department of Computer Science Korea National Open University

요 약

적응형 소프트웨어의 개발을 어렵게 만드는 가장 큰 원인은 환경의 다양함에 대응하기 위해 솔루션이 복잡해짐에 따라 소프트웨어의 개발과 유지보수 비용이 증가한다는 것이다. 이를 막기 위해 기존의 방법을 사용하여 솔루션을 환경 별로 분리한다 해도, 소프트웨어에 변경이 일어나면 분리된 솔루션 모두에 반영해 주어야 하기 때문에 그 비용이 실로 적지 않다. 이 논문에서는 솔루션 사이의 차이를 룰로 정의함으로써 기반 솔루션으로부터 다른 솔루션들을 자동으로 생성해내는 방법을 제안함으로써 위와 같은 문제를 해결하고자 하였다.

1. 서 론

적응형 소프트웨어의 목표는 “다양한 환경 속에서도 소프트웨어가 목표하는 바를 달성하는 것”[1]이라고 할 수 있다. 적응형 소프트웨어 개발이 일반적인 소프트웨어에 비해 어려운 이유는 대응해야 하는 환경의 수가 늘어날수록 솔루션의 복잡도가 증가하기 때문이다. 다음 예[2]를 보자.

```
(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (let ((wildcard (directory-wildcard dirname)))
    #+(or sbcl cmu lispworks)
    (directory wildcard)
    #+openmcl
    (directory wildcard :directories t)
    #+allegro
    (directory wildcard :directories-are-files nil)
    #+clisp
    (nconc
     (directory wildcard)
     (directory (clisp-subdirectories-wildcard wildcard)))
    #-(or sbcl cmu lispworks openmcl allegro clisp)
    (error "list-directory not implemented")))
```

이 코드는 디렉토리 안의 파일 목록을 보여주기 위한 커먼 리습(Common Lisp) 함수로서, 여러 가지의 리습 구현들(sbcl, cmu, lispworks, openmcl, allegro, clisp 등) 위에서 똑같이 작동하도록 작성된 것이다. #+와 #- 기호는 그 다음에 나오는 표현식이 참일 때 각각 더하고 빼야 할 코드들을 보여준다. 솔루션이 복잡하더라도, 이 코드와 같이 하나의 함수로 추상화하여 사용함으로써 코드를 간단하게 만들 수 있다. 하지만 개발 도중 더 나은 알고리즘을 알게 되어, 함수의 내용을 수정하게 되었다고 생각해보자. 그 경우, 각 환경에 해당하는 모든 코드를 다 고쳐야 한다. 또는 새로운 환경이 추가되었을 경우, 모든 함수에 새로운 환경을 다루는 코드를 추가해야만 한다. 또한 여러 가지 환경을 다루기 위해 분기문을 지나치게 사용한 코드는 읽기 어렵다. 환경에 대응하기 위한 곁가지들 속에서 줄기에 해당하는 부분을 가려 읽어야 하기 때문이다. 이와 같이 각각의 환경에 대응하는 부분들을 종합하여 하나의 솔루션으로 기술하는 방식은 솔루션 자체를 복잡하게 만들고, 적응형 소프트웨어의 개발과 유지보수에 드는 비용을 증가시킨다.

이와 같은 문제를 해결하기 위해서는, i) 각각의 환경에 대응하는 솔루션을 분리하고, ii) 환경을 파악하여 그에 맞는 솔루션을 적용하는 것이 필요하다. Context Oriented Programming (COP)[3], Aspect Oriented Programming[4], Feature Oriented Programming[5], 스트래티지 패턴(strategy pattern)[6] 등을 이용하여 솔루션들을 분리하고 문맥에 따라 여러 가지의 솔루션 중 하나를 선택하는 것이 가능하다. 하지만 기존의 방법들은 다음과 같은 한계를 가진다. 첫째, 솔루션의 분리가 객체나 함수 단위이기 때문에 라인 단위로 솔루션 간의 차이를 기술하는 것이 어렵고, 일부 언어에는 적용이 불가능하다. COP나 스트래티지 패턴은 객체 지향 언어에서만 사용 가능하다. 즉, CSS(Cascading Style Sheets)와 같은 언어에는 전혀 적용될 수 없다. 둘째, 대응해야 하는 환경의 수가 많을 경우 솔루션들의 변경에 따르는 비용이 크다. 소프트웨어에 대한 요구사항이 변하여 솔루션이 달성해야 하는 내용 자체가 변했을 경우, 분리한 각각의 솔루션들을 일률적으로 변경해 주어야 한다.

이 논문에서는 위와 같은 문제들을 해결할 수 있는 솔루션의 분리 방법을 기술하려고 한다. 뒤에 자세하게 설명하겠지만, 이상적인 방법은 기준이 되는 솔루션과 다른 솔루션 사이의 상대적 관계를 기술한 뒤, 하나의 솔루션만을 작성하고 다른 솔루션들은 자동적으로 생성되게 하는 것이다. 솔루션의 분리 방법에만 초점을 맞추는 이유는, 환경에 맞는 솔루션을 적용하는 일은 비교적 간단한 일이기 때문이다. 논문은 다음과 같이 구성된다. 2장에서 관련 연구를 살펴본 후, 3장에서 솔루션의 분리 방법에 대한 아이디어를 설명한다. 4장에서는 앞에서 기술한 아이디어를 CSS 코드의 사례에 적용해 보이고, 5장에서 결론을 맺는다.

2. 관련 연구

Context Oriented Programming(COP)이란 소프트웨어가 처하게 될 상황들을 각각의 레이어로 정의한 후, 각 레이어에 클래스와 메써드를 나누어 정의하고 동적으로 레이어를 활성화시킴으로써 소프트웨어가 상황에 맞는 행동을 하게 만드는 프로그래밍 방식을 말한다. 현재 커먼 리습, 스몰토크, 자바 용의 라이브러리가 나와 있다. 레이어간의 관계를 기술하는 방법은 아직 제공하고 있지 않다. COP에 레이어간 상속을 구현하고 부모 레이어에 포함된 클래스와 메써드를 자손 레이어에서 부분적으로 재정의할 수 있게 한다면, 본 논문에서 의도하는 바를 보다 조직화된 방식으로 달성할 수 있을 것이다. 향후에 연구해야 할 과제라고 하겠다.

Aspect Oriented Programming(AOP)은 소프트웨어의 여러 부분에서 공통적으로 나타나는 특성을 추상화 할 수 있는 방식을 제공한다. 어떤 부분에(pointcut) 어떤 코드를 추가할 것인지(advice)를 정의하면 전처리 과정에서 코드를 삽입하기 때문에 함수로 추상화하기 어려운 부분을 추상화 할 수 있고 인라인 단위의 삽입이 가능하다는 장점이 있다.

Feature Oriented Programming(FOP)은 컴퓨터의 사양을 고르고 주문하듯이 소프트웨어의 특성을 선택함으로써 소프트웨어를 생성하는 방법을 말한다. FOP에서는 컴파일 타임에 다양한 특성들의 조합을 선택함으로써 빌드되는 소프트웨어가 달라진다. AOP와 비슷한 면이 많으며, 최근에는 빌드 전에 특성들의 조합 결과를 정적으로 분석하는 도구도 구현되었다.

Model Driven Arichitecture[7]를 이용하면 정의한 모델과 환경별 프로파일로부터 환경에 맞는 코드를 생성해 낼 수 있다. 하지만 M이라는 모델과 P_a ... P_z 까지의 프로파일로부터 S_a ... S_z 까지의 스켈레톤 코드가 생성되었다 해도, 개발자는 각 코드에 모두 추가적인 작업을 해 주어야 한다. 모델이 완성된 코드를 만들어낼 수 있을 정도로 모든 정보를 담는 것은 불가능하기 때문이다. 그 모든 정보를 담고 있는 모델은, 이미 모델이 아니라 코드일 것이다. 따라서 이 논문에서는 모델이 아니라 하나의 베이스 코드와 코드들 사이의 variation으로부터 다른 코드들을 생성해내는 방식을 취하였다. 이와 같은 방식은 생성된 코드들에 추가적인 수작업이 필요 없다는 장점이 있다.

3. 적응형 소프트웨어 솔루션의 분리

적응형 소프트웨어의 정의에는 여러 가지가 있다. 적응 목적은 자가 치유, 기능 유지, 지속적인 개선 등이 될 수 있고, 이를 위한 소프트웨어 적응 방식도 파라미터 적응과 아키텍처 적응 등으로 나뉜다[8]. 하지만 이 모든 것의 공통점은 적응을 유발하는 원인이 환경이라는 것과 환경의 변화에도 불구하고 소프트웨어가 가진 본래의 목적을 최대한 달성하는 것이 중요하다는 것이다. 따라서 적응형 소프트웨어는 여러 가지 환경에 대처하기 위한 각각의 솔루션들의 종합이라는 형태로 구성된다. 그리고 여러 개의 솔루션을 종합해서 하나로 만든 솔루션은 단일한 솔루션보다 복잡하기 때문에 개발과 유지보수 비용이 커지게 된다. 다음 예[9]를 보자.

```
#sidebar {
    padding: 10px;
    border: 5px solid black;
    width: 230px;          /* for Internet Explorer */
    voice-family: "₩"3₩"";
    voice-family: inherit;
    width: 200px;          /* real value */
    }
html>body #sidebar {
```

```
    width: 200px;
    }                          /* for Opera */
```

이 코드는 인터넷 익스플로러와 오페라, 기타 브라우저에서 모두 똑같은 모양의 웹 페이지를 보여주기 위한 CSS 코드이다. 인터넷 익스플로러가 너비 값을 해석하는 방식의 차이 때문에 인터넷 익스플로러를 위한 너비값을 먼저 지정한 후, voice-family 속성을 사용해 인터넷 익스플로러가 선언이 끝난 것처럼 인식하게 하고, 그 후에 다른 브라우저를 위한 너비 값을 선언한 것이다. 하지만 오페라 브라우저도 인터넷 익스플로러처럼 voice-family 속성을 만나면 해석을 멈추기 때문에 아래 쪽에 오페라 브라우저만을 위해 추가적인 선언을 해 주었다. 이 코드는 서로 다른 브라우저 환경에서 동일한 모양을 보여준다는 목표를 달성하기 위해 일종의 CSS 트릭을 이용하고 있다. 이런 방식은 코드를 복잡하게 만들 뿐 아니라 브라우저들의 버전이 바뀌어 CSS를 해석하는 규칙이 변한다면 더 이상 유효하지 않게 된다. 하지만 CSS 기법을 담고 있는 많은 책들이 서로 다른 브라우저 문제를 해결하기 위해 이와 비슷한 트릭들에 의존하고 있다.

위 코드를 주석 없이 본다면 사실상 어떤 부분이 인터넷 익스플로러를 위한 부분이고 어떤 부분이 나머지 브라우저를 위한 부분인지, 맨 아래 오페라를 위해 삽입한 코드는 왜 들어가 있는건지, 알기가 어렵다. 하지만 각각의 CSS 코드를 분리해서 상황에 맞게 로드할 수 있다면 다음과 같이 될 것이다.

```
/* Internet Explorer */
#sidebar {
    width: 230px;
    padding: 10px;
    border: 5px;
    }

/* the others */
#sidebar {
    width: 200px;
    padding: 10px;
    border: 5px;
    }
```

이 편이 코드를 읽기는 훨씬 쉽다. 따라서 환경에 대응하는 각각의 솔루션을 분리하는 것이 복잡도를 줄이는 방법이다. 하지만 S라는 솔루션을 S_a, S_b, ... S_z으로 분리했을 경우, 요구사항이 변하거나 다른 해결 방법을 사용하고자 하여 S_a가 S_a'으로 변경되면 S_b, ... S_z 도 같이 변경해 주어야 한다는 번거로움이 생긴다.

따라서 이상적인 방법은 S_a와 나머지 솔루션 사이의 차이(variation)를 V_{ab}, V_{ac}, ... V_{az}로 정의한 다음, S_a만을 변경해 나가고 나머지 솔루션은 S_a에 V_{ab} ... V_{az}를 적용해 자동으로 생성하는 것이다.

솔루션 사이의 차이를 어떻게 기술할 수 있을까? 필요한 것은 코드에서 "관심의 대상이 되는 부분"을 정의하고 그 부분에 "규칙에 의한 변화"를 가해서 다른 솔루션을 만들어내는 것이다. AOP의 pointcut과 advice가 이와 비슷한 역할을 한다고 할 수 있다. 따라서 솔루션 사이의 차이를 정의하기 위해, pointcut과 advice 같은 개념들을 이용할 수 있다. 단, AOP는 횡적 관심(crosscutting concern)을 추상화하기 위해 pointcut과 advice를 사용하는 반면에, 여기서는 새로운 솔루션을 만들어내기 위해 사용한다는 점에서 그 의도가 다르다. 대부분의 언어가 AOP 관련 라이브러리를 가지고 있기 때문에, 기존 라이브러리를 수정하여 새로운 솔루션을 만들어내도록 하는 일은 비교적 용이할 것으로 생각된다.

4. 사례 연구

이 같은 생각을 웹 어플리케이션의 사례에 적용해 보았다. 다양한 브라우저에서 "동일하게 보이고 동일하게 동작하는" 웹 어플리케이션을 만들기란 쉽지 않다. 브라우저라는 환경이 너무나 다양한 탓이다. 다양한 환경을 다루는 솔루션은 단일한 환경을 다루는 솔루션에 비해 복잡해지게 된다. 적응형 소프트웨어를 만드는 것이 어려운 것과 같은 이유다. 사실, "다양한 환경에서 제대로 기능하는 것"이 적응형 소프트웨어의 목표라고 할 때, 웹 어플리케이션은 브라우저라는 다양한 환경에서 제대로 동작하려고 애쓰는 적응형 소프트웨어라고 할 수 있다. 브라우저의 다양성이 자바스크립트나 CSS 코드를 지저분하게 만드는 주 원인이다. 결코 언어 자체의 문제가 아닌 것이다. 앞 장에서 기술한 아이디어를 이 문제에 적용한다면 다음과 같이 할 수 있을 것이다. 하나의 브라우저에 대해서만 CSS나 자바스크립트 코드를 작성한 후 브라우저 사이의 차이를 정의한 룰을 적용해서 다른 브라우저들에 적합한 CSS와 자바스크립트 코드를 생성해 내는 것이다. 일단 각 브라우저에 적합한 코드들을 만들어 냈다면, 요청의 정보를 살펴보고 해당 브라우저에 맞는 코드를 로드하게 만드는 것은 쉬운 일이다. 구글 툴바 설치과정이 OS와 브라우저 정보를 바탕으로 요청을 처리하는 좋은 예라고 할 수 있다.

위와 같은 아이디어를 variation-applier[10]라는 도구로 구현해보았다. variation-applier는 Common Lisp으로 작성되었으며 weblocks[11] 웹 프레임워크 위에서 실행된다. 실행을 위해서는 Common Lisp과 weblocks를 설치하고, 소스 코드의 build-and-test.lisp 파일을 SLIME(The Superior Lisp Interaction Mode for Emacs)으로 로드(ctrl+c ctrl+l)한 후, SLIME에서 (variation-applier:start-variation-applier)를 치면 된다.

브라우저의 주소창에 localhost:8080 을 쳐 넣으면 입력 화면이 나타나는 것을 볼 수 있다.

이 도구는 CSS 파일을 대상으로 한다. 사용 방법은 기본 솔루션에 해당하는 디렉토리와, variation 룰들이 들어 있는 디렉토리, 솔루션들이 생성될 디렉토리를 지정하고 apply를 누르면, variation 룰 당 하나씩 새로운 솔루션이 생성된다. 그림 1 을 보면 base 디렉토리에 들어있는 6개의 CSS 파일에 대해서 variation 디렉토리의 IE.var와 Opera.var라는 variation rule이 적용되었고, result 디렉토리의 하위 디렉토리에 IE, Opera 디렉토리 및 CSS 파일들이 생성된 것을 볼 수 있다.

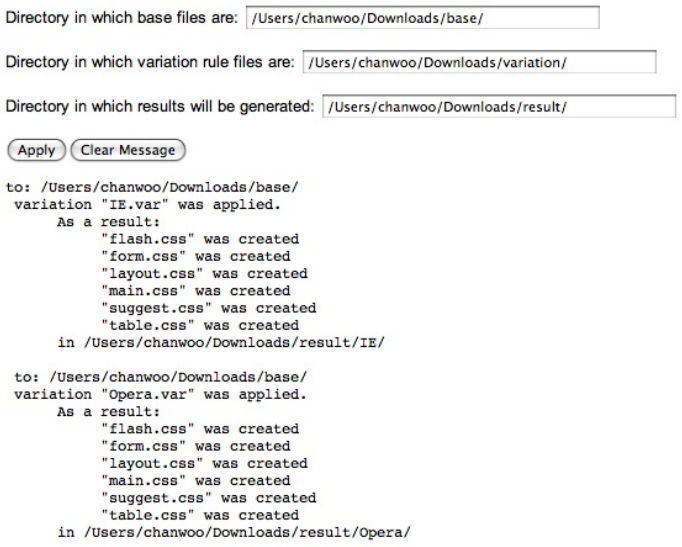

그림 1 variation-applier의 실행 화면

variation rule은 pointcut과 advice의 개념을 이용은 하되, 최대한 CSS 문법과 비슷하게 기술하도록 하였다. variation 문법은 다음과 같다.

```
rule-clause*
rule-clause::= selector-clause* { declaration-clause* }
selector-clause::= global | css-selector
declaration-clause::= property: value;
property::= css-property
value::= css-value | variation-list
variation-list::= (variation*)
variation::= (operator [property] {css-value | op-list})
operator::= before | around | after | delete | calculate
op-list::= {{+|-|*|/|} number [px|em|%]}*
```

variation rule은 다음과 같이 적용된다. 베이스 코드와 variation 코드의 CSS 선택자를 비교하여 선택자가 같거나 variation 쪽의 선택자가 global이면 variation rule을 적용한다. CSS 선언 블록 내의 각각의 선언을 비교하여 property가 같은 선언일 경우 variation-list를 적용한다. variation-list에 들어있는 variation의 오퍼레이터가 before나 after일 경우에는 해당 선언의 앞 뒤에 variation의 property와 value로 이루어진 새로운 선언을 추가한다. around일 경우에는 해당 선언 대신 variation의 선언을 삽입하고, delete일 경우에는 베이스 코드의 선언을 없앤다. calculate의 경우에는 variation에서 선언한 연산을 베이스 코드의 선언의 value에 차례대로 적용한다.

예를 들면 다음과 같다.

```
/* base file */
#sidebar
{ width: 200px; margin: 0;}

case 1
/* var file */
#sidebar
{ width: ((calculate + 30px)); }
=>
/* result file */
#sidebar
{ width: 230px; margin: 0; }

case 2
/* var file */
#sidebar
{ width: ((before border 5px) (calculate * 1.1 + 10) (after
padding 10px)); }
=>
/* result file */
#sidebar
{ border: 5px; width: 230px; padding: 10px; margin: 0;}

case 3
/* var file */
global
{ width: ((around border 0)) margin: ((delete)) }
=>
/* result file */
#sidebar
{ border: 0; }
```

3장에서 들었던 예를 variation-applier 를 이용해 분리한다면 다음과 같이 표현할 수 있을 것이다.

```
/* base file */
#sidebar {
    width: 200px;
    padding: 10px;
    border: 5px;
```

```
    }

/* IE.var file */
#sidebar {
    width: ((calculate + 30px));
      }
```

단지 하나의 선언에 대해서라면 맨 처음 예와 비교하여 코드 몇 줄이 줄었을 뿐이지만, 선언의 개수가 많고, 개발 과정에서 코드를 변경하는 경우가 잦을수록 variation-applier를 이용함으로써 절감할 수 있는 비용은 커질 것이라고 쉽게 예상할 수 있다.

variation-applier를 이용하면 CSS 핵이나 필터를 이용하지 않고도 브라우저들의 CSS 해석 문제를 해결할 수 있다. 핵은 원하는 방법대로 브라우저를 동작시키려고 쓰는 일종의 임시방편이라고 할 수 있는데, 브라우저에 따라 다른 결과를 얻기 위해서 필터를 사용한다[12]. 필터는 특정 브라우저에 어떤 규칙을 적용시키거나 적용시키지 않기 위해서 브라우저의 버그나 적합하지 않은 CSS 규칙을 사용하는 것을 말한다. 핵과 필터의 문제는 브라우저간 환경의 차이를 극복하기 위해 브라우저의 버그나 버전에 특화된 성질에 의존하는 경향이 있다는 것이다. 이를 이용하면 차후에 브라우저의 새 버전이 나와서 버그가 없어질 경우, 제대로 동작할지를 보장할 수 없게 된다. 또한 이와 같은 문제 외에도, 핵과 필터를 사용하면 CSS 유효성 검사를 통과하는 코드를 만들어 내기가 어렵게 된다. variation-applier 역시 특정 버전에 대해 기술된 규칙에 따라 솔루션을 만들어 내지만, 해당 솔루션은 그 버전에만 적용되므로, 브라우저의 새로운 버전이 나온다고 해서 제대로 동작하지 않게 될 가능성은 없다. 또한 CSS 규칙을 준수하는 브라우저를 베이스 솔루션으로 개발하고 버그를 가진 브라우저들을 위한 솔루션은 variation 규칙을 통해 생성해 낸다면, 베이스 솔루션을 CSS 유효성 검사의 대상으로 삼아 통과 여부를 알아보기도 쉽다.

핵이나 필터를 이용한 코드를 variation-applier를 적용하여 개선하는 예를 보이도록 하겠다. 윈도우용 인터넷 익스플로러 6 이하 버전에서는 자식 선택자를 지원하지 않기 때문에 IE 6 이하 버전에서 배경의 투명 이미지를 보이지 않게 하고 싶을 경우, 자식 선택자 필터를 이용하여 다음과 같이 할 수 있다[12].

```
html>body { background-image: url(image.gif); }
```

이 경우에는 코드를 읽는 사람이 IE 6 이하 버전에서 자식 선택자를 무시한다는 것을 모른다면 배경 이미지가 나타나지 않는다는 것을 예측하기 어렵다. variation-applier를 이용한다면 IE5.var 또는 IE6.var 파일에 다음과 같이 선언하면 된다.

```
body { background-image: ((delete)); }
```

이 경우에는 각 버전의 브라우저에서 해당 선언이 적용되지 않을 것임을 쉽게 알 수 있다.

또한 IE 6 이하에서 나타나는 버그로 플로트 엘리먼트 근처에 텍스트가 있을 때 텍스트 주변에 3 픽셀의 공간이 생기는 버그가 있다[12]. 이 버그를 해결하는 기존 방법은 홀리 핵이라고 불리는 핵을 사용하여 텍스트 엘리먼트에 높이와 마진을 주고 플로트 엘리먼트에는 -3 픽셀의 오른쪽 마진 값을 주는 것이다. 원래 코드가 다음과 같다고 하면,

```
.float { fload: left; width: 100px; }
p { margin-left: 100px; }
```

홀리 핵을 사용한 코드는 다음과 같다.

```
* html p { height: 1%; margin-left: 0; }
* html .float { margin: 0 -3px; ma₩rgin: 0;}
```

이 코드의 margin 사이에 나오는 역슬래시는 오타가 아니라 IE 5.x 버전과 IE 6.x 버전 사이의 차이를 해결하기 위한 핵을 사용한 것이다. 매우 지저분한 해결 방법이라고 할 수 있다. variation-applier를 이용한 코드는 다음과 같다.

```
/* IE 5.var */
p { margin-left: ((before height 1%)
                  (around margin-left 0)); }
.float { float: ((around margin 0 -3px));
        width: ((delete)); }

/* IE 6.var */
p { margin-left: ((before height 1%)
                  (around margin-left 0)); }
.float { float: ((around margin 0));
        width: ((delete)); }
```

핵과 필터를 알지 못해도 각 버전 별로 어떤 코드를 의도하고 있는지가 매우 명확하게 드러나는 것을 볼 수 있다.

5. 결 론

적응형 소프트웨어 개발의 어려움이 다양한 환경에 따른 솔루션의 복잡성 때문임을 밝히고, 개발과 유지보수 비용을 줄일 수 있는 솔루션의 분리 방법을 제시하였다. 또한 웹 어플리케이션 영역을 예로 들어

CSS 코드를 대상으로 하는 솔루션 분리 도구를 구현하였다. 향후에는 솔루션 사이의 차이를 라인 단위로 기술하는 것에 그치지 않고 COP를 확장하여 보다 조직적으로 기술하도록 만들 수 있을 것이다. 또한 CSS보다 전문적인 프로그래밍 언어에 대해 이와 같은 방법을 적용하여 유효성을 알아보는 것도 필요할 것이다.

6. 참고 문헌

[1] R. Laddaga, "Self Adaptive Software Problems and Projects," *Proc. of the Second International IEEE workshop on Software Evolvability*, pp. 3-10, Sep. 2006.

[2] P. Seibel, *"Practical Common Lisp,"* Apress, 2005.

[3] R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-oriented Programming," Journal of Object Technology, Vol. 7, No. 3, pp. 125-151, Mar.-Apr., 2008.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," *Proc. of the 15th European Conference on Object-Oriented Programming*, Vol. LNCS 2072, pp. 327-353, Springer, June, 2001.

[5] D. Batory, "Feature Oriented Programming and the AHEAD Tool Suite," *Proc. of the 26th International Conference on Software Engineering*, 2004.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software,"* Addisson-Wesley, 1994.

[7] S. Mellor and M. Balcer, *"Executable UML: A foundation for model-driven architecture,"* Addison-Wesley, 2002.

[8] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "Composing Adaptive Software," *Computer*, Vol. 37, No. 7, pp. 56-64, July, 2004.

[9] D. Cederholm, *"Web Standard Solutions,"* friends of ED, 2004.

[10] http://onlisp.blogspot.com/2008/03/blog-post_27.html

[11] http://common-lisp.net/project/cl-weblocks/

[12] A. Budd, S. Collison, and C. Moll, *"CSS Mastery: Advanced Web Standards Solutions,"* friends of ED, 2004.