논·문

클래스분석 시 예외분석의 필요성*

이정수, 조장우
부산외국어대학교 컴퓨터공학과

Necessity of Exception Analysis at Class Analysis

Jung-Su Lee, Jang-Wu Jo
Department of Computer Engineering,
Pusan University of Foreign Studies

요약

이론적으로 자바의 예외분석과 클래스분석은 상호 의존적인 관계이다. 이로 인해 클래스분석 시 예외분석을 동시에 수행하게 되면 실용적인 분석기 개발이 어렵다. 그러나 클래스분석 시 예외분석의 의존도가 낮다면 두 분석을 분리함으로서 보다 실용적인 분석기를 설계할 수 있다. 이 논문에서는 두 분석이 상호의존적으로 필요한 경우를 제시하고, 자바 프로그램들을 대상으로 이러한 경우가 빈번하게 발생하는지에 대한 분석을 수행한다.

1. 서론

이론적으로 정확한 예외분석을 위해서는 클래스분석 정보를 필요로 하고, 또한 정확한 클래스분석을 위해서는 예외분석 정보를 필요로 한다. 그러므로 정확한 클래스 분석을 위해서는 예외분석을 동시에 수행해야 한다. 예외분석과 클래스분석의 시간 복잡도는 N이 분석 단위의 수일 때 각각 $O(N^3)$이다.[6,7] 이러한 복잡도로 인해 두 분석을 동시에 수행하는 것은 실용적이지 못하다고 알려져 있다[7]. 그러나 클래스분석과 예외분석의 상호 의존적인 경우가 많이 발생하지 않는다면, 클래스분석과 예외분석을 분리함으로서 보다 실용적인 분석기를 설계할 수 있다.

이 논문에서는 두 분석이 상호의존적으로 필요한 경우를 제시하고, 자바 프로그램들을 대상으로 이러한 경우가 빈번하게 발생하는지에 대한 분석을 수행한다.

* 본 연구는 한국과학재단 목적기초연구 (2000-1-30300-009-2) 지원으로 수행되었음.

이 논문의 구성은 다음과 같다. 2 장에서는 클래스분석과 예외분석의 상호 관계를 알아보고, 3 장에서는 실제 자바 프로그램을 대상으로 두 분석이 의존적으로 필요하게 되는 사용빈도를 알아본다. 4 장에서는 실험을 통해 얻은 결과를 토대로 결론을 기술한다.

2.클래스분석과 예외분석의 상호관계

클래스분석이란 어떤 식이 가리키는 객체의 타입(클래스)을 정적으로 유추하는 분석이다[1,6,7,8]. 자바에 대한 집합-기반 클래스분석은 [1]에 기술되어 있다. 예외분석이란 각 식에 대해서 실행 중 발생될 수 있는 예외들을 정적으로 유추하는 분석이다.

그림 1은 클래스분석 시 예외분석 정보가 필요한 경우와, 예외분석 시 클래스분석 정보가 필요한 예이다. 우선 정확한 예외분석을 위하여 클래스분석 정보가 필요한 경우는 다음과 같다. 그림 1의 try 블록내의 메소드 호출문 e_1.m()에서 발생 가능한

```
try {
    …
     e1.m();
} catch (Exception e) {
        …
        throw e;
        e.m();
}
```

그림 1 예외분석과 클래스분석이 상호 의존적인 예

예외는 호출된 메소드 m()에서 처리되지 않고 전달된 예외들을 포함한다. e_1의 타입이 T이고 m()이 T의 하위 클래스들에서 재정의 되어 있는 경우, 자바는 동적인 바인딩을 지원하므로 T와 T의 하위 클래스들의 메소드 m()이 수행될 수 있다. 그러므로 e_1.m()에서 발생 가능한 예외는 T와 T의 하위 클래스들의 m()에서 발생 가능한 예외들을 포함한다. 그러나 e_1의 클래스분석 정보를 이용하면 수행 가능한 메소드가 전체 재정의된 메소드들 중에서 일부이므로, 예외분석의 정확도를 향상시킬 수 있다.

다음으로 정확한 클래스분석을 위하여 예외분석 정보가 필요한 경우는 다음과 같다. 그림 1의 catch 절에서 catch 인수 e를 사용하는 경우인 throw e와 e.m()을 보자. e의 클래스분석 결과는 try 구문 안에서 발생 가능한 예외들 중에서 Exception 클래스 또는 이의 하위 클래스이다. try 블록에서 발생 가능한 예외들은 예외분석 정보이므로, 클래스분석 시 예외분석이 필요한 경우이다. 이와 같은 이유로 예외분석과 클래스분석이 병행되어야만 각각의 분석이 보다 정확해 질 수 있다.

그림 2 는 예외분석과 클래스분석을 동시에 수행하는 분석을 집합 기반 분석으로 설계하였다. 프로그램의 모든 구문 e 에 대해 두 종류의 집합 관계식이 생성된다. $X_e \supseteq se$는 클래스분석을 위한 집합 관계식이고, $P_e \supseteq se$는 예외분석을 위한 집합

[New] $> \text{new } c : \{\chi_e \supseteq \{c\}\}$

[This] $> \text{this} : \{\chi_e \supseteq \{c\}\}$ if c is the enclosing class

[FieldAss] $$\frac{> e_1 : C_1}{> id.x := e_1 : \{\chi_{c.x} \supseteq \chi_{e_1} \mid c \in \chi_{id}\} \cup \{\chi_e \supseteq \chi_{e_1}, P_e \supseteq P_{e_1}\} \cup C_1}$$

[Seq] $$\frac{> e_1 : C_1 \quad e_2 : C_2}{> e_1 ; e_2 : \{\chi_e \supseteq \chi_{e_2}, P_e \supseteq P_{e_1} \cup P_{e_2}\} \cup C_1 \cup C_2}$$

[Cond] $$\frac{> e_0 : C_0 \quad e_1 : C_1 \quad e_2 : C_2}{> \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \{\chi_e \supseteq \chi_{e_1} \cup \chi_{e_2}, P_e \supseteq P_{e_0} \cup P_{e_1} \cup P_{e_2}\} \cup C_0 \cup C_1 \cup C_2}$$

[FieldVar] $$\frac{> id : C_{id}}{> id.x : \{\chi_e \supseteq \chi_{c.x} \mid c \in \chi_{id}\} \cup C_{id}}$$

[Throw] $$\frac{> e_1 : C_1}{> \text{throw } e_1 : \{P_e \supseteq \chi_{e_1} \cup P_{e_1}\} \cup C_1}$$

[Try] $$\frac{> e_0 : C_0 \quad e_1 : C_1 \quad e_2 : C_2}{> \text{try } e_0 \text{ catch } (c_1\, x_1\, e_1)(c_1, x_2, e_2) : \{\chi_e \supseteq \chi_{e_0} \cup \chi_{e_1} \cup \chi_{e_2}, \chi_{x_1} \supseteq P_{e_0} \cap C_1 *, \chi_{x_2} \supseteq P_{e_0} \cap C_2 *\} \cup \{P_e \supseteq P_{e_0} - \{c_1, c_2\} * \cup P_{e_1} \cup P_{e_2}\} \cup C_0 \cup C_1 \cup C_2}$$

[MethCall] $$\frac{> e_1 : C_1 \quad e_2 : C_2}{> e_1.m(e_2) : \{\chi_e \supseteq \chi_{e_2}\} \cup \{\chi_e \supseteq \chi_{c.m} \mid c \in \chi_{e_1}\} \cup \{P_e \supseteq P_{c.m} \mid c \in \chi_{e_1}\} \cup C_1 \cup C_2}$$

[MethDef] $$\frac{> e_m : C}{> \text{c.m} : \{\chi_{c.m} \supseteq \chi_{e_m}, P_{c.m} \supseteq P_{e_m}\} \cup C}$$

[ClassDef] $$\frac{> \text{class } c = \{\text{var } x_1, \ldots, x_k, m_1, \ldots, m_n\} \quad m_i : C_i (1 \le i \le n)}{> \quad c : C_1 \cup \wedge \cup C_n}$$

[Program] $$\frac{> c_i : C_i,\ i = 1, \ldots, n}{> p : C_1 \cup \wedge \cup C_n}$$ if a program $p = c_1, \ldots, c_n$

그림 2 예외분석을 병행하는 클래스 분석의 설계

관계식이다. X_e 는 클래스들을 위한 집합-변수이고, P_e 는 처리되지 않는 예외들을 위한 집합-변수이다.

그림 2의 생성규칙들 중에서 예외분석과 클래스분석이 상호 의존적인 try-catch 문에 대한 생성규칙을 살펴보자. $X_{x1} \supseteq Pe_0$ 부분과 같이 x_1의 클래스분석을 위하여 예외분석 정보인 Pe_0 가 필요한 정보이다.

그러나 클래스분석과 예외분석을 동시에 수행하게 될 때의 시간 복잡도는 $(2N)^3$ 이다. 여기서 N은 프로그램의 식, 문장 그리고 메소드의 수이다. [7]에서 이러한 클래스분석이 이론적으로는 정확한 분석결과를 제공하지만, 분석의 복잡도로 인해 실용적이지 못하다는 결과를 제안하였다.

그러므로 실제 프로그램에서 두 분석이 의존적으로 작용하는지를 살펴보고자 한다. 만약 그 의존성이 상대적으로 적다면 정확도가 감소되지 않고 두 분석을 분리하여, 각 분석을 효율적으로 수행하는 것이 가능하다.

3. 실험 및 분석

본 절에서는 소스 코드가 공개되어 있고 자주 사용되는 자바 프로그램들을 대상으로 예외와 관련된 구문들을 조사해보고, catch 절 인수의 사용형태, catch 절 인수의 사용 빈도 등을 조사해본다. 그리고 대상 프로그램에서 예외분석과 클래스분석을 동시에 수행해야 하는 소스 코드가 있는지를 살펴보고, 그 분석의 결과 타입이 명시적인 catch 절 인수의 타입과 얼마나 차이가 나는지를 비교 분석해본다.

3.1 대상 프로그램의 소개와 특징

3.1.1 대상 프로그램의 간단한 소개

표 1은 실험에 사용된 프로그램들로서 약 70 KByte ~ 3.8 MByte 의 다양한 크기의 자바 프로그램들이다. 대상 프로그램들의 구문적 특징을 보여주기 위해, 각 프로그램의 클래스, 인터페이스, 그리고 메소드의 수를 보여주고, 예외 관련 구문들의 사용빈도를 보여주기 위해서, try, catch, 그리고 throw 구문들의 수를 보여준다.

속성 / 이름	Class수	Interface 수	Method 수	Try수	Catch수	Throws 수	KBytes
jasmin[8]	11	0	77	2	4	39	107.0
JavaSim [9]	29	0	207	16	18	83	72.6
jb[10]	52	0	515	19	21	116	154.3
java_cup [11]	45	0	333	9	9	115	315.7
sablecc [12]	280	4	1753	78	77	11	612.3
jel[13]	47	11	385	100	107	57	341.2
JFlex[14]	51	4	395	20	26	31	461.5
antlr[15]	156	31	1886	76	87	292	1,047.8
mork [16]	144	9	1332	48	59	280	708.9
comice.tar [17]	10	2	127	17	21	40	88.3
tomcat [18]	189	15	2033	209	248	353	1,232.0
jasper [19]	98	16	703	80	91	309	591.9
jess[20]	235	11	1085	115	148	436	548.9
soot[21]	1063	182	8923	33	41	61	3,764.1

표 1 실험 대상 프로그램의 간략한 특징

3.1.2 대상 프로그램의 예외와 관련된 특징

표 2는 실험에 사용된 프로그램들의 예외와 관련된 구문적인 특징을 보여준다. **Catch절 안의 throw** 는 catch 절 안에서 사용되는 throw 구문의 개수이고, **Catch절 밖의 throw** 는 catch 절 밖에서 사용되는 throw 구문의 수이다. 표 2에서 throw 구문은 catch 절에 관계없이 사용됨을 알 수 있다.

throw e*의 타입**에서 **변수**는 *throw e*에서 *e*가 변수로 사용된 경우의 수이고, **메소드 호출**은 *e*가 메소드 호출로 사용된 경우의 수이다. 그리고 **객체 생성**은 *e*가 ***new를 사용한 객체 생성인 경우의 수이다. 표 2에서 98% 이상이 새로운 예외 객체를 생성하여 예외를 발생함을 알 수 있다.

3.2 catch 인수의 사용 형태와 사용 빈도

catch 절의 인수가 catch 절 내에서 사용되는 형태를 살펴보면, catch 절의 인수 역시 reference 타

속성 / 이름	throw 수		catch 인수로 throw	catch 인수로 메소드 호출	throw e의 타입		
	catch절 안의 throw	catch절 밖의 throw			변수	메소드 호출	객체 생성
jasmin	2	16	0	3	0	0	18
JavaSim	1	36	0	0	0	0	37
jb	11	42	1	1	1	0	52
java_cup	1	27	1	6	1	0	27
sablecc	58	21	2	2	2	0	77
jel	6	38	2	11	2	1	41
JFlex	9	49	0	3	0	0	58
antlr	25	300	10	18	10	0	315
mork	35	284	6	16	33	0	286
comice.tar	1	16	0	17	0	0	17
tomcat	39	102	18	95	31	0	110
jasper	50	149	9	23	12	0	187
jess	87	113	21	47	32	14	154
soot	18	514	2	8	2	0	530
계	343	1,707	72	250	126	15	1,909

표 2 실험 대상 프로그램의 예외와 관련된 특징

입의 변수이므로 자바에서의 다른 타입의 객체와 마찬가지로 다양한 형태로 사용되어 질 수 있다. 대상 프로그램에서 catch 절의 인수가 사용되는 형태는 메소드의 인수, 배정문, 타입 캐스팅, 필드 접근, throw *e*, 메소드 호출 등이다. 이들 중에서 예외 분석이 필요한 경우는 throw *e*, 메소드 호출과 같은 형태로 사용된 것들이다.

catch 절의 인수 역시 Throwable 클래스 또는 그 하위 클래스의 타입이므로 catch절 안에서 명시적으로 예외를 발생시키는 throw 구문에 사용될 수 있다. 이와 같은 형태의 사용 빈도는 표 2의 **catch 인수로 throw** 필드에 나타난 것과 같다. 전체 throw 중에서 0.28%만 catch 인수를 사용함을 알았다.

그리고, catch 절의 인수 역시 예외 클래스 타입의 객체이므로 메소드 호출에 사용될 수 있다. 이러한 경우와 같이 사용된 빈도는 표 2의 **catch 인수로 메소드 호출** 필드에 나타난 것과 같다. 이런 경우는 대상 프로그램들 중에서 250번 나타났다.

3.3 catch 인수를 사용한 예에서 두 분석간의 의존성 분석

catch 인수를 throw 또는 메소드 호출에 사용하더라도, catch 인수에 대한 클래스분석 시 예외분석 정보가 항상 필요한 것은 아니다.

이번 실험에서는 catch 인수를 사용하는 예들에서, 클래스분석 시 예외분석이 의존적인 경우에 대한 분석을 한다.

catch 절의 인수에 명시된 객체의 타입과 실제 try 절의 예외분석 결과를 비교한 결과가 표 3과 표 4 이다. catch 절의 인수를 throw 구문에서 사용되는 경우는 표 3에 나타내고 catch 절의 인수가 메소드를 호출하는 형태로 사용될 경우는 표 4로

속성 / 이름	catch 절안의 throw	catch 인수로 throw					
		Number		type = value		type ≠ value	
jasmin	2	0	0%	0	0%	0	0%
JavaSim	1	0	0%	0	0%	0	0%
jb	11	1	9.1%	1	9.1%	0	0%
java_cup	1	1	100%	0	0%	1	100%
sablecc	58	2	3.4%	2	3.4%	0	0%
jel	6	2	33.3%	2	33.3%	0	0%
JFlex	9	0	0%	0	0%	0	0%
antlr	25	10	40%	8	32.0%	2	8.0%
mork	35	6	17.1%	6	17.1%	0	0%
com.ice.tar	1	0	0%	0	0%	0	0%
tomcat	39	18	46.2%	15	38.5%	3	7.7%
jasper	50	9	18%	7	14.0%	2	4.0%
jess	87	21	24.1%	18	20.7%	3	3.4%
soot	18	2	11.1%	2	11.1%	0	11.1%
계	343	72	21.0%	61	17.8%	11	3.2%

표 3 catch 절의 인수로 throw

나누어 나타내었다.

표 3에서 **Number** 필드는 catch 절 인수가 catch 절 내의 throw 구문에서 사용된 빈도를 나타낸다. **type = value** 필드는 catch 절에 명시된 예외 객체의 타입과 예외분석의 결과가 같은 타입인 경우의 수를 나타내고, **type ≠ value** 필드는 서로 다른 타입인 경우의 수를 나타낸다.

표 4에서 **Number** 필드는 catch 절의 인수가 메소드 호출로 사용되는 경우의 빈도를 나타낸다. **Not Overriden MethodCall** 필드는 호출되는 메소드가 하위클래스에서 재정의 되지 않는 경우이다.

3.4 실험 결과의 분석

표 3에서 **catch 절 안의 throw** 는 프로그램에서 catch

절 내에 throw 구문이 사용된 수이다. **Number** 필드에 있는 비율은 catch 절 안에 throw 구문이 있는 모든 경우와 catch 절의 인수가 catch 블록 속의 throw에서 발생되는 예외의 타입으로 사용되었을 경우의 비율을 나타낸다. **type = value** 에 나타나는 비율은 catch 절 안에 throw 구문이 있는 모든 경우(표 2의 Catch절 안의 throw 필드)와 catch 절의 인수의 타입이 예외분석의 결과가 같은 경우의 비율을 나타내고, **type ≠ value** 에 나타나는 비율은 catch 절 안에 throw 구문이 있는 모든 경우(표 2의 catch절 안의 throw 필드)와 catch 절의 인수의 타입이 예외 분석의 결과가 서로 다른 경우의 비율을 나타낸다.

표 3에서 catch 블록 안에서 사용된 전체 throw 개수 343 중에서 약 3.2 %인 11개만이 catch 절 인수와 다른 타입을 가지는 예외를 발생시킨다는 것을 알 수 있다. 각 프로그램마다 예외분석이 필요한 경우는 0에서 3개 정도의 적은 수로 필요로 함을 알 수 있다.

속성 / 이름	catch 절안의 메소드 호출	catch 인수를 메소드 호출에 이용							
		Number		type = value		type ≠ value			
						Overriden Method Call		Not Overridden Method Call	
jasmin	8	3	37.5%	3	37.5%	0	0%	0	0%
JavaSim	2	0	0.0%	0	0.0%	0	0%	0	0%
jb	2	1	50.0%	1	50.0%	0	0%	0	0%
java_cup	14	6	42.9%	4	28.6%	0	0%	2	14.3%
sablecc	60	2	3.3%	2	3.3%	0	0%	0	0%
jel	162	11	6.8%	8	4.9%	0	0%	3	1.9%
JFlex	23	3	13.0%	1	4.3%	0	0%	2	8.7%
antlr	139	18	12.9%	10	7.2%	3	2.2%	5	3.6%
mork	58	16	27.6%	6	10.3%	0	0%	10	17.2%
com.ice.tar	26	17	65.4%	13	50.0%	0	0%	4	15.4%
tomcat	240	95	39.6%	55	22.9%	0	0%	40	16.7%
jasper	56	23	41.1%	16	28.6%	0	0%	7	12.5%
jess	137	47	34.3%	41	29.9%	0	0%	6	4.4%
soot	36	8	22.2%	8	22.2%	0	0%	0	0%
계	963	250	26.0%	168	17.4%	3	0.3%	79	8.2%

표 4 catch 절의 인수로 메소드 호출

표 4의 **catch 절안의 메소드 호출** 필드는 catch 블록 속에서 메소드를 호출하는 모든 경우의 수를 나타낸다. 표 4에서 나타난 모든 비율들도 catch 블록 속에서 메소드를 호출하는 모든 경우의 수에 대한 비율들이다.

표 4에서 **type ≠ value**인 경우 중에서 **Not Overriden MethodCall** 필드에 나타난 것과 같이 제정의 되지 않은 메소드를 호출한다는 것은 실제 예외분석을 수행해서 나온 결과는 catch절 인수의 하위 타입에 있는 메소드지만, 실제로 호출되는 메소드는 catch 절에 명시된 인수의 타입에 있는 메소드를 호출한다는 것이다. 그러므로 이러한 메소드들에 해당되는 catch 절은 예외분석의 결과가 catch 절에 명시된 인수의 타입과 같다고 할 수 있다.

여기서 catch 절에 명시된 인수의 타입이 예외 분석의 결과와 다르게 사용된 비율은, **type ≠ value** 경우 중에서 **Overriden MethodCall** 필드에 나타난 것과 같이 antlr 프로그램에서 3개 뿐인데, 이것은 catch 블록 내에서 사용한 총 963개의 메소드 호출 구문 중에서 약 0.3% 정도임을 알 수 있다. 그러나 여기서 사용된 메소드는 Throwable 클래스에서 상속받은 사용자 정의 예외 클래스의 toString() 이다. Throwable 클래스 및 사용자 정의 예외 클래스의 toString() 메소드의 명세를 살펴봐도 예외가 발생되지 않음을 확인 할 수 있다. 그러므로 catch 절의 인수의 타입과 예외분석을 수행한 타입이 거의 일치함을 알 수 있다.

4. 결론

이론적으로 정확한 예외분석을 위해서는 클래스분석 정보를 필요로 하고, 또한 정확한 클래스분석을 위해서는 예외분석 정보를 필요로 한다. 그러나 클래스분석과 예외분석의 상호 의존도가 낮다면 클래스분석과 예외분석을 분리함으로서 보다 실용적인 분석기를 설계할 수 있다.

본 논문에서는 자바 프로그램들을 대상으로 두 분석간의 의존적인 경우에 대한 분석을 수행하였다.

클래스분석과 예외분석이 상호 의존적으로 필요한 경우는 catch 절의 인수의 타입이 catch 블록 내에서 메소드 호출을 할 경우와 throw 구문의 예외 객체로 사용될 경우이다.

본 논문에서 클래스분석 시 예외분석의 정보를 필요로 하는 경우가 아주 낮은 비율로 두 분석간의 의존도가 적다는 것을 알 수 있었다. 그러므로 정확도가 많이 감소되지 않고 두 분석을 분리하여, 각 분석을 효율적으로 수행하는 것이 가능하다는 것을 알 수 있었다.

참고문헌

[1] B.-M. Chang, K. Yi, and J. Jo, "Constraint-based analysis for Java," SSGRR2000 Computer and e-business Conference, August 2000, L'Aquila, Italy.

[2] B.-M. Chang, J. Jo, K. Yi, and K. Choe, "Interproceduaral Exception Analysis for Java," In Proceedings of 2001 ACM symposium on Applied Computing, pages 620-625, Mar. 2001

[3] K. Yi and B.-M. Chang, "Exception analysis for Java," 'ECOOP'99 Workshop on Formal Techniques for Java Programs, June 1999.

[4] K. Arnold and J. Gosling, "The Java Programming Languages, Second Edition," Addison-Wesley, 1997

[5] N. Heintze, "Set-based program analysis," Ph.D thesis, Carnegie Mellon University, Oct, 1992.

[6] J. Dean, D. Grove, and C. Chambers, "Optimization of objected-oriented program using static class hierarchy," In Proceedings of 9th European Conference on Objected-Oriented Programming, pages 77-101, Aug. 1995

[7] G. Defouw, D. Grove, and C. Chambers, "Fast interprocedural class analysis," in Proceedings of 25th ACM SIGPLAN Symposium on Principles of Programming Languages, pages 222-236, Jan. 199i.

[8] F. Tip and J. Parlsberg, "Scalable Propagation-Based Call Graph Construction Algorithms," In Proceedings of 15th Annual Conference on Objected-Oriented Programming Systems, Languages, and Applications, pages 281-293, Oct. 2000

[9] http://mrl.nyu.edu/~meyer/jvm/jasmin.html

[10] http://javasim.ncl.ac.uk

[11] http://www.cs.colorado.edu/serl/misc/jb.html

[12] http://www.cs.princeton.edu/~appel/modern/java/CUP

[13] http://www.sablecc.org

[14] http://galaxy.fzu.cz/JEL

[15] http://jflex.sourceforge.net

[16] http://www.antlr.org

[17] http://mork.sourceforge.net

[18] http://www.ice.com/java/tar

[19] http://www.jboss.org

[20] http://www.jboss.org

[21] http://herzberg.ca.sandia.gov/jess

[22] http://www.sable.mcgill.ca/soot

이정수

1997년~2001년 부산외국어대학교 컴퓨터공학과(학사)

2001년~현재 부산외국어대학교 컴퓨터공학과 석사과정

조장우

1987년~1992년 서울대학교 계산통계학과(학사)

1992년~1994년 서울대학교 전산과학과(석사)

1994년~현재 한국과학기술원 전산학과 박사과정

1997년~현재 부산외국어대학교 컴퓨터공학과 조교수

관심분야는 프로그램분석, 객체지향프로그래밍, 프로그래밍 언어