반응적 에이전트 프레임워크를 위한 패턴 언어
(A Pattern Language for the Reactive Agent Framework)

박 성 운 † 정 재 민 †† 박 수 용 †††
(Sung-Woon Park) (Jae-Min Jeong) (Soo-Yong Park)

요 약 최근 몇 년간 소프트웨어 에이전트가 소프트웨어 공학의 새로운 추상화 단위로서 연구되고 있다. 자율성, 적응성, 협력성 등의 속성을 갖는 에이전트는 특히 분산 시스템, 오픈 시스템, 복잡한 시스템 등의 영역에서 시스템을 구성하는 새로운 추상화 단위로서 이해되고 있다. 그러나 에이전트에 관한 개념과 특성에 관한 연구가 꾸준히 진행되고 있음에도 불구하고, 에이전트 패러다임을 위한 프로그래밍 언어의 실용화는 요원한 상태이다. 이는 에이전트가 갖는 속성이 문제 영역별로 매우 다양해서 이러한 모든 속성을 공통으로 만족시키는 구현 방법에 대한 공통된 의견이 도출되지 못하고 있기 때문이다. 이러한 현실에서 프로그래밍 언어보다 한 단계 상위 계층에 존재하는 프레임워크를 통한 에이전트 개발은 보다 현실적인 대안으로 받아들여지고 있다. 그러나 에이전트 프레임워크를 개발하기 위한 많은 기술들은 개발자간에 공유되고 있지 않아서 동일한 시행착오를 반복하는 경우가 많다. 본 논문에서는 이러한 문제의 해결책으로 객체지향 기술에서 주로 사용되는 패턴 언어를 사용함으로써, 에이전트 프레임워크 개발의 경험과 지식을 개발자간에 공유하고자 한다. 본 논문은 반응적 에이전트 프레임워크의 개발을 위한 패턴 언어를 제안하고 ATAM[1]에 기반한 검증의 절차를 거쳤다. 이러한 간접 경험의 증가는 반복해서 발생하는 시행착오를 감소시킴으로써 개발자들이 보다 본질적인 문제에 집중할 수 있도록 도와준다. 이는 결국 고품질의 에이전트 프레임워크 개발에 기여할 것으로 기대된다

키워드 : 에이전트, 프레임워크, 패턴, 패턴 언어

Abstract Recently software agent has been studied as a new abstraction unit of software engineering. The agent with autonomous, adaptability and cooperation attribute is accepted as a new abstraction unit especially in distributed systems, open systems, and complex systems. However, the progress of agent research has been slow and the realization of agent programming language seems to be far distant. Because the properties of agent are diverse, the opinions of researchers can not converge to one. In this situation, software agent framework is accepted more realistic alternative solution. However the knowledge for its development doesn't have been shared among developers. So they often have to make same errors. We will help sharing of knowledge and experience by using pattern language which has been used in object technology for long times. This paper proposes a reactive agent framework pattern language and validates it based on ATAM[1] The increase of such indirect experience can reduce the waste of resource by preventing the same try and error. So agent framework developers are able to concentrate on more essential issues. Finally quality of software agent framework will be increased.

Key words : Agent, Framework, Pattern, Pattern Language

· 본 연구는 한국과학재단 목적기초연구 R01-2003-000-10197-0 지원으로 수행되었음
· 본 연구는 정보통신부 지원 ITRC 프로그램의 지원을 받아 수행되었음
† 비 회 원 : 아이티플러스
lazz@selab.sogang.ac.kr
†† 비 회 원 : 서강대학교 컴퓨터학과
jmjeong@selab.sogang.ac.kr
††† 정 회 원 : 서강대학교 컴퓨터학과 교수
sypark@ccs.sogang.ac.kr
논문접수 : 2003년 1월 8일
심사완료 : 2003년 11월 21일

1. 서 론

가트너 그룹은 자율적인 소프트웨어 에이전트와 인공지능 소프트웨어를 포함하는 기업 자동화가 10년 내에 전체 정보 기술 분야의 50%를 차지할 것이라고 예측했다. 2010년까지 이는 약 2500억 US 달러의 가치가 있을 것이라고 한다[2]. 그 외에도 에이전트는 사용자에게 특화된 인터페이스, 전자 상거래, 엔터프라이즈 통합, 제조업, 비즈니스 지원, 통신, 네트워크 관리 등 다양한 분

야에서 각광받기 시작하고 있다[3].

이러한 성장에도 불구하고 에이전트 기술의 실현은 많은 난항을 겪고 있다. 그 원인 중 하나는 구현 하부구조의 미성숙이다. 지금까지 객체지향, 컴포넌트 등 새로운 패러다임의 대중화는 항상 그 구현기술의 확산과 함께 이루어졌다. 현재 에이전트 분야에서의 구현 기술은 객체지향과는 달리 프로그래밍 언어가 아닌 프레임워크를 중심으로 발전하고 있다. 이는 에이전트가 갖는 속성이 문제 영역별로 매우 다양해서 이러한 모든 속성을 공통으로 만족시키는 패러다임에 대한 공통된 의견이 도출되지 못하고 있기 때문이다. 이러한 현실에서 프레임워크의 성숙도를 높이는 것은 에이전트 기술의 대중화를 위한 현실적 대안으로 받아들여서 지금은 FIPA [4], MASIF[5] 등의 에이전트 프레임워크를 위한 표준이 만들어져 사용되고 있다.

이를 보다 가속화 시키기 위한 한가지 방법은 에이전트 프레임워크 개발자들의 경험과 지식을 서로 공유하는 것이다. 프레임워크는 그 특성상 자신을 이용해 구축될 애플리케이션들이 어떤 특성을 가지고 있는지, 어떻게 진화해 갈 것인지를 예측할 수 있어야 한다. 그리고 이는 매우 선험적인 작업이므로 많은 경험과 시행착오를 통해서만 올바른 결과를 도출할 수 있다. 이러한 간접 경험의 증가는 반복해서 발생하는 시행착오를 감소시킴으로서 개발자들이 보다 본질적인 문제에 집중할 수 있도록 도와준다. 이는 결국 고품질의 에이전트 프레임워크 개발에 기여할 것으로 기대된다

현재 대중적으로 널리 사용되는 설계 경험의 공유 방법은 패턴과 패턴 언어이다. 반면 패턴의 등장 이전에 이를 위한 전통적인 방법은 오픈소스였다. 이는 이미 GNU등의 활동을 통해 그 성과가 입증되었다. 이 방법은 에이전트 프레임워크에서도 현재 널리 쓰이고 있다 [1,7,8]. 그러나 오픈 소스 코드는 개발자의 수많은 고민의 최종 결과만을 담고 있기 때문에 설계 시 발생하는 여러 트레이드오프와 설계 순서와 같은 과정의 역추적은 종종 불가능하다. 하지만 패턴은 그 선택의 이면에 있는 이유와 이론적인 배경, 선택의 결과와 구현 상의 이슈를 모두 다루고 있기 때문에 타인이 그 경험을 적절히 변형해 사용하기에 보다 효율적이고 손쉽다[9]. 패턴 언어는 그런 패턴들을 연속적으로 적용해서 보다 큰 문제를 해결하기 위한 순서의 조합을 제시하기에 보다 커다란 문제를 해결하기에 유용하다.

본 연구에서는 지금까지 소스 코드라는 베일에 싸여 있던 에이전트의 내부 설계를 패턴의 형태로 다시 표현해서 에이전트 프레임워크 개발자들의 경험을 보다 효율적으로 공유할 수 있는 방법을 제시하고 그 연구 방법의 한 사례를 제시했다. 그리고 반응적 에이전트의 내부 설계 부분에 대한 경험을 제시해서 다른 에이전트 프레임워크 개발자들의 간접 경험을 증가시키는 데 공헌했다.

2. 관련연구

본 논문과 직접적인 관련을 갖는 연구는 에이전트 프레임워크와 에이전트 아키텍처의 연구가 있다. 에이전트 프레임워크로서는 FIPA-OS[6]와 JADE가 대표적이다. 위 두개의 프레임워크는 Agent간의 통신과 상호작용의 아키텍처에 초점을 맞추고 있고, 제품의 특성상 성능, 가용성 등의 품질에 있어서는 좋은 가이드를 해주고 있으나 오픈 소스로서 효율적인 정보 공유가 어렵다.

보다 구체적으로 논하자면, FIPA-OS는 FIPA Agent의 표준안을 첫 번째로 구현한 Open Source 프로젝트로서, Agent Communication, Agent Management, Agent Message Transport, Agent Software Integration의 아키텍처를 제공한다[6]. JADE는 Java Agent DEvelopment Framework의 약자로서, 자바 언어로 FIPA- Spec을 구현한 것이다[7]. 위 두 개의 프레임워크는 Agent간의 통신과 상호작용의 아키텍처 즉 상호운용성에 초점을 맞추고 있다. 그러나 소스 코드만이 공개되어 있어 내부 설계에서 어떤 트레이드 오프가 존재했는지를 유추하는 것이 불가능하다. 또한 FIPA의 방침은 에이전트의 내부 구조에 대해서는 표준화하지 않는 것이 기본 전제이다[12].

본 논문과 유사한 관점으로 에이전트 프레임워크에 접근한 연구로서는 Kendal의 에이전트 프레임워크[11]가 있다. Kendal은 여러 가지 객체지향 디자인 패턴을 이용하여 계층적 구조를 갖는 에이전트 프레임워크를 제안하였다. 하나의 에이전트는 계층 아키텍처 패턴(layer architectural pattern)을 이용하여 구성되는 데, 각 계층은 지각(sensory), 자기 정보(belief), 추론(reasoning), 행위(action), 협력(collaboration), 중계(translation), 이동(mobility)의 작업으로 세분화된다. 이와 같은 계층 구조의 장점은 계층 간 종속성이 줄어들고, 계층 별 재사용이 가능한 이점이 있는 반면, 효율성이 떨어지고 불필요한 작업이 수행될 수 있다[3]. 사실 이 같은 특징은 Kendal의 에이전트가 갖고 있다기보다 계층 아키텍처 패턴 자체의 장단점으로 볼 수 있다.

3. 반응적 에이전트 프레임워크 패턴 언어

본 연구에서는 반응적 에이전트 프레임워크를 개발하기 위한 패턴 언어를 제시한다. 에이전트 프레임워크와 패턴 언어의 특성으로 인해 본 논문에서는 문제 영역과 설계의 범위를 한정시켰다. 먼저 연구의 효율성을 위해

문제 영역은 반응적 에이전트로 국한시켰다. 반응적 에이전트란 동적으로 변화는 환경에 자신의 행동을 바꾸는 에이전트를 뜻한다[13]. 이는 프레임워크의 특성상 문제 영역을 한정했을 때 설계시 트레이드오프의 선택을 분명히 할 수 있기 때문이다. 그리고 설계의 범위는 다중 에이전트 아키텍처를 배제한 에이전트 아키텍처로 한정했다. 에이전트 내부 모델로도 불리는 에이전트 아키텍처는 하나의 에이전트가 모듈 집합의 구성으로 어떻게 분해될 수 있는가, 그리고 이러한 모듈들의 상호작용의 정의이다[12]. 범위를 에이전트 아키텍처로 제한한 이유는 현재 다중 에이전트 시스템의 표준인 FIPA에서 다중 에이전트 아키텍처의 표준을 제시하고 있는 반면 에이전트 아키텍처에 대한 부분은 개발자 임의로 하도록 규정하고 있기 때문이다.

3.1 반응적 에이전트 프레임워크가 추구하는 품질 속성

본 논문은 개발하고자 하는 프레임워크의 중요 품질 속성을 결정하기 위해 소프트웨어의 노화(Software Aging)라는 속성에 주목했다[15]. 소프트웨어의 노화란 소프트웨어가 개발된 이후로 지속적인 유지 보수 비용을 유발시키는 특성으로, 시간이 경과함에 따라 소프트웨어의 수정 비용 역시 증가함을 말한다. 소프트웨어 제품의 유지보수 비용이 개발 비용의 4배 정도가 됨을 감안하면[16] 이는 반드시 주목해야 할 특성이다[17]. 이러한 소프트웨어의 특성을 고려할 때 프레임워크는 다음과 같은 품질 속성이 중요하다.

첫 번째, 프레임워크의 성공을 위한 조건은 쉽게 유지보수할 수 있어야 하는 능력이다. 급변하는 요구사항을 반영하지 못하면 도태되기 때문이다. 하지만 이는 프레임워크의 경우 애플리케이션보다 획득하기 어려운 품질이다. 프레임워크는 재사용이 목적이기 때문에 이를 사용한 모든 애플리케이션과의 호환성을 고려한 버전업을 해야 하기 때문이다. 이를 위해서는 초기에는 나중에는 확보하기 어려운 품질 요소 또한 후의 변화에 유연하게 대처할 수 있는 능력을 갖추어야 한다. 즉 성능과 보안과 같은 제품의 품질 속성보다는 이식성과 변경용이성 등의 속성을 우선적으로 확보해야 한다.

두 번째, 프레임워크는 애플리케이션이 품질 속성을 보다 쉽게 획득할 수 있도록 지원할 수 있는 능력이 요구된다. 프레임워크는 독립적으로 사용되는 것이 아니라 애플리케이션과 결합되어서 존재하기 때문에, 그 최종형태-애플리케이션과 결합된 형태-의 품질이 가장 중요하기 때문이다. 애플리케이션은 성능, 보안, 사용편이성 등의 품질 속성에 의해서 평가 받는다[18]. 반면 프레임워크는 애플리케이션이 그러한 품질 속성을 보다 쉽게 획득할 수 있도록 가이드할 수 있는 능력이 더욱 중요하다[17].

이 두 가지 성공 요인을 획득하는 것은 매우 도전적인 작업이다. 프레임워크는 해당 문제 영역의 여러 애플리케이션들의 공통점과 차이점을 파악해서 각각 콜드 스팟(cold spot) 과 핫 스팟(hot spot)을 구성해야 하는데, 이는 매우 선험적인 작업이기 때문이다[5]. 완벽한 선험성은 사실상 불가능하므로 제품의 초기에는 많은 시행착오를 통해 학습을 하게 된다. 그런 경험을 통해 올바른 추상화와 관계를 추출해 낼 수 있고, 최종적으로 적절한 아키텍처가 만들어진다.

이러한 사실에 근거해서 본 연구에서 추구한 품질 속성은 반응적 에이전트 애플리케이션과 프레임워크의 유지보수를 용이하도록 지원하는 능력이다. 초기에 확보하기 어려운 변경 용이성라는 품질 속성을 확보할 수 있는 방법을 제시함으로서 향후 그 프레임워크와 애플리케이션이 원하는 다른 품질을 확보하는데 도움이 되리라 기대한다.

3.2 접근 방법

본 연구는 [19]의 방법을 응용해서 프레임워크를 개발한 후, 이로부터 패턴 언어를 추출하였다. [19]에 의하면 프레임워크는 그림 1과 같이 발전해 나간다. 먼저 3개의 예제를 통해 경험을 점차적으로 습득해 나가면서 그로부터 프레임워크를 만들어 나간다. 만들어 나가는 과정은 상속 위주의 화이트 박스, 컴포지션 위주의 블랙박스 등 여러 단계를 통해 발전을 거친다. 이런 과정을 거치는 이유는 프레임워크는 경험으로부터 도출될 때 그리고 적용 결과에 피드백을 받아 그 경험과 함께 발전할 때 성공할 수 있기 때문이다[19].

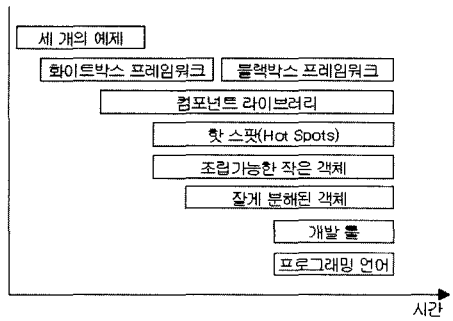


그림 1 프레임워크의 발달 과정

본 연구에서는 이 방법을 응용해 그림 2와 같은 방법을 사용했다. 먼저 3개의 예제 애플리케이션 중 하나로 신호등 관리 에이전트를 만들었다. 그리고 나머지 2개의 예제 애플리케이션 대신 JADE와 FIPA-OS를 분석했다. 이 3개 예제로부터 얻은 교훈을 토대로 프레임워크를 개발했다. 동시에 이 프레임워크를 이용해서 신호등

관리 에이전트를 리팩토링(Refactoring)했다. 이 애플리케이션에 적용하면서 얻은 새로운 교훈은 다시 프레임워크에 피드백을 주었다. 이렇게 두 가지 경험-애플리케이션 개발, 프레임워크 개발-은 서로에게 영향을 주면서 발달해갔다. 이 작업들을 통해 얻은 경험 중 현재 완성되어 있는 에이전트 아키텍처 부분을 패턴 언어로 추출하였다. 마지막으로 이 패턴 언어가 확보하는 품질 속성을 검증하고 보다 정확히 표현하기 위해 ATAM[1]을 이용해 평가하였다.

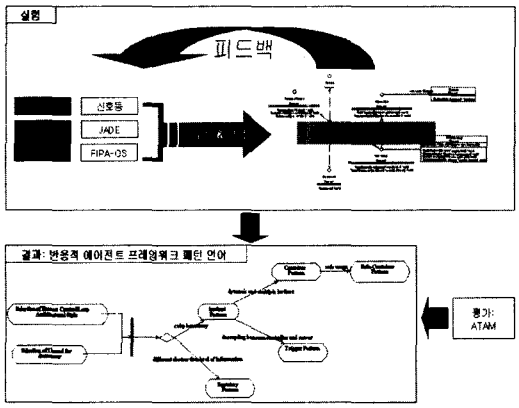


그림 2 접근 방법

그림 1과 그림 2를 연결해 본 논문에서 직접 개발한 부분과 그렇지 않은 부분을 정확히 기술하면 표 1과 같다.

이렇게 경험 중심의 연구 방법을 선택 한 이유는 연역적으로는 패턴 언어를 만들 수 있는 방법이 없기 때문이다. 캡슐화, 정보 은폐 등의 개념은 추상적이어서 위 개념으로부터 직접적이면서 구체적인 해결책인 패턴을 찾기 어렵다. 따라서 실제 개발 없이 추측에 의해 올바른 추상화를 이끌어 내는 것은 인간의 사고 방식으로는 대부분 실패한다[19]. 실제로 패턴은 항상 누군가가 새로운 설계 방식을 발명해서 제안하는 것이 아니라 실제 개발에서 발생한 문제를 발견하는 방식으로 세상에 알려진다.

3.3 신호등 관리 에이전트 예제

본 논문은 프레임워크 개발을 위해 예제를 개발하는 과정을 기반으로 하였다. 그 예제는 신호등 관리 에이전트로서 다음과 같은 시나리오로 동작한다. 시나리오에서 본 논문이 다루고 있는 에이전트 아키텍처는 1, 2, 3에 해당한다. 4번 시나리오는 다중 에이전트 아키텍처에 해당하는 내용으로 본 논문의 범위를 벗어나는 내용이다.

1. 4거리의 4개 신호등을 모두 관리하는 하나의 에이전트가 있다.
2. 에이전트는 지속적으로 교통량의 변화를 측정한다.
3. 특정 방향으로 진행하는 차량의 숫자가 정해진 임계치를 넘을 경우, 정해진 규칙에 의해 신호등의 점등 주기를 변경한다.
4. 이러한 변화는 인접한 4개 에이전트에게 통보되고, 서로의 목표가 위배될 경우 공동선의 도출을 위해 협상이 이루어진다.

3.3 반응적 에이전트 프레임워크 패턴 언어

패턴 언어의 기술 방법은 [20]의 형식을 따른다. 본 기술 방법의 목적은 반응적 에이전트 프레임워크의 아키텍처를 설계하기 위해 개별적인 패턴들을 독립적으로 사용하는 것 이상의 효과를 내기 위해서 패턴들을 연속적으로 사용하는 방법을 정의하는 것이다.

본 연구의 성과인 패턴 언어를 UML의 Activity Diagram으로 표현하면 그림 3과 같다. 이 그림은 각 패턴들의 적용 순서와 관계에 대해 기술하고 있다. 여기에서는 먼저 각 패턴들에 대해 하나하나 패턴기술 형태에 맞춰 살펴본다. 그 다음에 각 패턴들의 연결 관계에 대해 논한다.

참고로 본 패턴언어에서 Instinct 패턴, Container패턴, Safe-Container패턴은 같이 연동해 상호작용을 보이므로 Safe Container 패턴에서 일괄적으로 상호작용을 묘사하겠다.

가. Controller, Sensor, Actuator Architectural Pattern

동기: 에이전트 아키텍처를 구성하기 위해 처음 결정해야 할 트레이드오프는 모델의 선택이다. 현재 에이전트 아키텍처의 모델은 BDI[21], CSA[22]등 시간을 통해 검증된 여러 가지가 존재한다. 각 모델은 서로간에 장단점이 있으므로 개발하고자 하는 프레임워크가 목적하는 문제 영역의 특성에 맞게 잘 선택해야 한다.

본 논문이 선택한 에이전트는 Nwana의 분류법 중

표 1 본 논문의 연구 범위

프레임워크의 발달 [그림 1]	본 논문의 응용 [그림 2]	본 논문의 연구 영역
세 개의 예제	JADE, FIPA-OS, 신호등 관리 에이전트	소스코드 분석: JADE, FIPA-OS 예제 개발: 신호등 관리 에이전트
화이트박스 프레임워크	Ants Framework	예제 개발
컴포넌트 라이브러리, 핫스팟, 조립가능한 작은 객체	Ants Framework 내의 인터페이스와 클래스	예제 개발

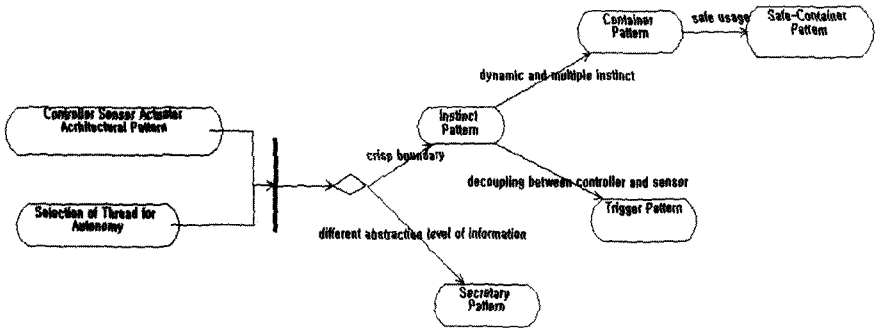


그림 3 반응적 에이전트 프레임워크 패턴 언어

반응적 에이전트에 해당한다[13]. 에이전트는 그 추상화 단위가 객체보다 큰 만큼[23] 각 기능들이 명확한 경계선을 구성할 수 있도록 분해할 수 있는 가이드가 필요하다. 이에 대해 지금까지 BDI, DESIRE 등 여러 방법이 사용되어 왔다[10]. 이들 중 반응적 에이전트에서 널리 쓰인 방법 중 하나인 Process Control Loop 아키텍처 스타일을 적용한 Control/Sensor/Actuator 모델[20]을 선택했다.

이름: CSA, Controller/Sensor/Actuator

의도: 실행 시간에 동적인 외부 환경의 변화를 감지 반응하여 외부 환경에 영향을 주는 에이전트를 구현하기 위한 적절한 크기의 모듈화 방법을 제공한다.

구조:

클래스 이름: *Controller*	
책무:	상호작용:
에이전트의 행동을 결정한다. Actuator에게 명령을 내린다.	Actuator

클래스 이름: *Sensor*	
책무:	상호작용:
외부환경을 지속적으로 감지하여 Controller에게 정보를 전달한다.	Controller

클래스 이름: Actuator	
책무:	상호작용:
Controller로부터 받은 명령을 외부에 수행한다.	

Sensor는 에이전트가 외부 환경을 인식하는 수단이다. Sensor는 외부 환경을 지속적으로 감지한다. 이 때 감지된 정보는 Controller에게 전달되어 Controller의 판단의 자료가 된다. 신호등 예제에서는 각 차선의 교통량이 *Controller*에게 전달되는 정보이다.

Controller는 에이전트의 지능을 담당한다. Sensor로부터 받은 데이타를 토대로 에이전트의 행동을 결정해서 그 결정 사항을 Actuator에게 전달한다. 신호등 예제에서는 각 차선별 교통량을 토대로 하여 어느 차선에 신호를 많이 할당할 것인가를 결정하는 부분이다. 에이전트를 지능적으로 설계할수록 Controller 컴포넌트는 점점 비대해진다. 그럴 경우에는 *Controller*를 또 다시 분해하거나 추가 컴포넌트와 결합시킬 수 있다. 예를 들어 *FIPA-OS*에서는 에이전트에게 지능을 부여하기 위해서 JESS 프레임워크를 이용하기도 한다.

Actuator는 에이전트가 외부 환경에 영향을 미치는 수단이다. Controller로부터 전달 받은 명령을 외부에 전달하는 전령의 역할을 수행한다. 이 때 외부는 실제 외부 환경이 될 수도 있고 타 에이전트가 될 수도 있다. 신호등에서는 각 차선별로 신호등의 신호 시간을 변경함으로써 외부환경에 영향을 미친다.

상호작용: Sensor는 지속적으로 주변 환경을 감지해서 그 내용을 *Controller*에게 전달한다. *Controller*는 그 내용을 기초로 해서 판단을 내리고 그 결과를 Actuator를 통해 주변 환경에 반영시킨다.

결과: 외부 환경을 지속적으로 센싱하고 그에 반응해야 하는 반응적 에이전트의 특성을 고려한 일반적인 아키텍처를 제공한다. 세부 설계와 달리 아키텍처는 나중에 변경하는 것이 어렵다. 또한 이후 발생할 변화를 정확히 예측하는 것은 현실적으로 불가능하다. 이 두 가지 모순점을 해결하는 방법은 검증된 아키텍처를 사용하는 것이다. Process Control Loop은 오랫동안 여러 분야에서 쓰여 왔고, 대부분 반응적 에이전트와 유사한 문제 영역에서 사용되어졌다. CSA 모델은 반응적 에이전트의 초기 분할을 위한 검증된 아키텍처를 제공한다.

CSA 중에서 지능과 관련되어 있는 기능은 Controller에 지역화되어 있다. 지능은 에이전트를 구현할 때 가장 비대해지기 쉽고 다른 기능들과 별도로 진화하기 쉬운 고려 사항이다. 이러한 고려사항을 분리시켰기 때문에 *Controller* 컴포넌트와 다른 컴포넌트들을 별도로 변화시키는 것이 가능할 뿐만 아니라, Controller의 내부에 *BDI* 모델과 같은 다른 모델을 적용하는 것 또

한 용이하다.

나. Secretary Design Pattern

동기: CSA 모델을 적용하면서 처음으로 봉착한 문제는 Sensor와 Actuator가 다루는 정보와 Controller가 다루는 정보의 수준이 다르다는 것이다.

신호등 관리 에이전트 예제에서 보자. Sensor가 감지하는 정보는 [북쪽 신호등: 현재 좌회전 대기 차량이 18대, 직진 대기 차량 30대]와 같은 현실세계로부터 추출한 가공되지 않은 정보이다. Actuator도 마찬가지로 [북쪽 신호등: 직진 신호를 현재 1분에서 1분 10초로 증가시켜라] 와 같은 가공되지 않은 정보이다. 반면에 Controller가 필요로 하는 정보는 [북쪽 신호등: 평상시보다 좌회전 대기 차량 10% 증가, 직진 대기 차량 35% 증가]와 같은 지능적인 판단에 직접적으로 필요한 가공된 정보이다.

네트워크 프로그래밍에서 암호화/복호화하는 모듈을 별도로 분리하듯이, CSA모델에서도 이 세 개의 컴포넌트 이외에 별도로 컴포넌트간 정보를 연계하는 컴포넌트가 필요하다. 왜냐하면 정보의 추상화 수준을 중계해주는 모듈이 CSA 중 어딘가에 있으면 그 컴포넌트는 2개 이상의 concerns을 가지고 있기 때문에 고려사항의 분리(Separation of Concerns)가 올바로 이루어지지 않는다. 예를 들어 Controller가 그 기능을 가지고 있다고 하면, Controller는 Sensor가 다루는 정보의 추상화 수준까지 파악하고 있어야 하는 것이 된다. 그렇게 되면 그 컴포넌트는 cross-cutting concerns를 소유하게 되는 것이고 이는 그 컴포넌트의 진화에 큰 장애가 된다[24].

이름: Secretary

의도: 취급하는 정보의 수준이 다른 두 컴포넌트의 사이를 연계하면서, 각 정보를 상대 컴포넌트에게 필요한 형태로 가공한다.

구조:

클래스 이름: Secretary	
책무:	상호작용:
Sensor가 가진 정보를 가공해서 Controller에게 전달한다. Controller가 내린 명령을 Actuator에게 필요한 포맷으로 변형해서 Actuator에게 전달한다.	Controller Actuator

상호작용: Controller가 나머지 두 컴포넌트-Sensor와 Actuator-와 정보를 주고 받을 때는 항상 Secretary를 거쳐서 통신한다.

결과: 고려사항이 다른 모듈을 별도의 컴포넌트로 분리함으로서 작업 할당이 매끄럽다. 기존의 Secretary를 상속해서 새로운 Secretary를 정의하는 방식을 채택함으로서 손쉽게 Secretary와 다른 컴포넌트들이 독립적으로 진화할 수 있다. 실행시간에 Secretary를 다른 Secretary로 교체하는 것이 가능하다.

Credits: 이 패턴은 AutoPilot의 Data Reduction과 그 도입 의도가 같다[25].

다. Instinct Design Pattern

동기: Process Control Loop 아키텍처 스타일은 본래 에이전트를 위한 것이 아니기 때문에, 거기서 유래된 CSA 모델은 에이전트의 모든 책무를 할당 받기에 명확한 경계선을 구성하지 못한다. 실제로 신호등 관리 에이전트 예제에서 그런 기능으로서 다음과 같은 시나리오를 발견해 내었다.

1. 교통량의 변화가 미리 정의된 임계치를 넘지 않으면, 정해진 시간 간격으로 신호를 바꾼다.
2. 주기적으로 자신이 관리하는 지역의 교통량을 인접한 에이전트에게 알려준다

이 두개의 책무는 CSA의 어딘가에 할당하기에 적절하지 않다. 왜냐하면 CSA의 각 컴포넌트는 변화하는 동적환경에 반응하는 것을 기본 목적으로 하고 있기 때문이다. 위의 시나리오는 모두 기계적으로 정해진 작업을 반복 수행하는 기능들이다. 현재의 분할 방식으로 Controller에 할당하는 것이 가장 적합하지만, 지능적 판단을 내리는 두뇌라는 Controller의 본래 목적에는 어긋난다. 하지만 Controller는 CSA중 가장 복잡한 부분이다. 가능하면 지능적인 부분과 그렇지 않은 부분을 분리하는 것이 이후의 변화에 대처하기 용이하다.

따라서 에이전트를 구성하는 컴포넌트들이 명확한 경계선을 구성하도록 하기 위해 CSA 이외에 새로운 추상화를 필요로 했다. 마치 객체지향에서 절차적인 기능들을 모아서 유틸리티 클래스라고 이름 짓듯 반응적 에이전트의 설계에서도 그러한 단위가 필요했다.

이름: Instinct

의도: 반응적 에이전트의 특성과 무관해서 마치 본능처럼 지속적으로 수행되어야 하는 하나의 작업을 캡슐화 할 수 있는 추상화 단위를 제공한다.

구조:

클래스 이름: *Instinct*	
책무:	상호작용
지능적으로 수행될 필요가 없는 작업을 자신의 제어권내에서 Controller와는 별도로 수행한다.	Sensor Actuator

상호작용: Controller가 Instinct를 inner class로 정의해서 쓰레드로서 시작시킨다. Smalltalk과 같이 코드 블록을 매개변수로 전달하는 것이 가능한 언어라면 그러한 방법도 적절하다. 필요하다면 수행 중에 Instinct는 Sensor와 Actuator를 사용할 수 있다. 하지만 Sensor와 Actuator는 Instinct를 사용할 수 없다. 그럴 필요가 있다면 그 책무는 Controller에 있는 것이 올바르다.

결과: CSA 의 세 컴포넌트와 함께 명확한 경계선을 만들 수 있도록 도와준다. 각 컴포넌트가 명확히 하나의 고려사항만을 가지고 있는 것은 cross-cutting concerns가 없음을 의미하고, 이는 곧 각 컴포넌트가 미래의 변화에 유연하게 대처하면서 변경될 수 있음을 뜻한다[18].

라. Container Design Pattern

동기: 하나의 에이전트는 여러 개의 Instinct를 소유할 수 있다. 또한 인간과 달리 소프트웨어 에이전트는 새로운 Instinct를 실행시간에 추가, 삭제해야 하는 경우가 종종 생긴다. 신호등 관리 에이전트 예제에서 하나의 신호등 에이전트는 2개의 Instinct를 소유하고 있어야 했다. 때로는 주변 상황이 변하면 실행시간에 새로운 Instinct를 추가하거나 삭제하는 것도 필요하다. 다음 시나리오를 보자.

rush hour에 사용하는 Instinct와 평소에 사용하는 Instinct가 서로 다르다. 시간대별로 Instinct를 교체할 수 있어야 한다

이런 여러 개의 Instinct들이 하나의 에이전트에 존재하고 이들의 생성과 소멸을 관리하기에 적절한 모델이 Container 모델이다. Container 모델은 새로운 패턴은 아니다. EJB, DCOM, CCM 등 기존의 컴포넌트 모델들은 모두 이 Container 모델을 따른다. 컨테이너란 자바 웹서버가 서블릿을 호스팅하듯이, 하나의 컴포넌트 인스턴스를 호스팅하면서 관리하는 인스턴스이다. 컨테이너의 관리 범위는 적용 기술마다 다르지만 일반적으로 컴포넌트의 생성과 소멸, 디스크 풀링, 트랜잭션과 보안 서비스 등 비즈니스 로직을 보조하기 위한 하부구조 서비스들을 포함한다.

이름: Container

의도: 여러 개의 Instinct를 실행시간에 추가, 삭제하는 등의 관리를 한다.

구조:

클래스 이름: Container	
책무	상호작용
Instinct들의 생명주기를 관리한다. Controller가 Instinct의 생명주기에 관여할 수 있는 인터페이스를 제공한다. Controller가 Instinct를 추가할 수 있는 인터페이스를 제공한다.	Container가 속한 에이전트 내에 있는 모든 Instinct

하나의 에이전트에 하나의 컨테이너가 있고 그 안에 여러 개의 Instinct가 있다. 컨테이너의 조정은 Controller가 한다.

Credits: Container라는 개념은 MS의 OLE를 시작으로 컴포넌트 기술에서 널리 사용되어 왔다. 본 패턴은 그 Container를 Instinct의 특성에 맞추어 변형하였다.

결과: Instinct가 주로 비즈니스 로직을 다룬다면 Container는 주로 동시성 관리와 같은 기술적인 문제 즉 하부구조 서비스를 다룬다. 비즈니스 로직과 하부구조 서비스의 분리는 RPC를 비롯한 여러 기술에서 많이 행해져 왔다. 이 두 가지 책무의 분리는 여러 가지 효과를 가져왔다. 그 둘의 고려사항이 다르고 변하는 요구사항의 주기가 다르기 때문에 각각이 별도로 발달하는데 큰 도움을 주었다. 또한 Container를 통해 Instinct를 실행시간에 추가, 삭제가 가능했다. 그리고 구현 시 개인의 능력에 따른 작업의 분배가 용이하다.

마. Safe-Container Design Pattern

동기: Container와 Instinct만을 이용해 코딩을 하면 null pointer 문제가 발생하는 경우가 있을 수 있다. 개발자의 실수로 Instinct를 Container에 설정 안 한 상태에서 Container를 통해 Instinct를 조작하는 경우이다. 분명히 애플리케이션 개발 당시의 실수이다. 하지만 프레임워크 수준에서 이런 실수를 미연에 방지할 수 있도록 한다면 최종 애플리케이션의 품질은 그만큼 향상될 것이다.

이런 문제가 발생하게 된 근본적인 원인은 Container가 Instinct를 미세하게 조정할 수 있는 인터페이스를 Controller에게 제공하기 때문이다. 그 미세한 조정은 Controller가 Instinct를 관리할 수 있는 강력한 능력을 제공하지만 그 반대급부로 Controller는 버그가 발생하기 쉬운 코드를 만들 가능성이 높다.

Safe-Container 패턴의 해결방법은 프레임워크에서 정해진 규칙대로만 사용하도록 강제한 Container의 하위 클래스를 만든 것이다. [26]의 Template Method 패턴을 적용해서 호출 순서를 미리 정의하고 null 체크와 같은 검사가 실행 중에 반드시 먼저 수행되도록 한 것이다. 프레임워크에서 버그가 발생하기 쉬운 작업을 검사해주는 것은 개발자의 공격적 프로그래밍을 지원할 수 있을 것으로 기대한다[27].

이름: Safe-Container

의도: Container와 Instinct의 지나치게 미세한 조정 기능 제공으로 인해 개발자는 실수를 할 여지가 있다. 이를 미연에 방지하도록 올바른 사용법 대로만 코딩할 수 있는 수단을 제공한다.

구조:

클래스 이름: Safe-Container	상위 클래스: Container
책무:	상호작용:
미리 정의한 방법으로만 Controller가 Container를 사용하도록 필요한 규칙을 정하고, 그 규칙을 어길 경우 컴파일 에러 또는 실행시간 예외를 발생시킨다.	Safe-Container가 속한 에이전트 내에 있는 모든 Instinct

상호작용:

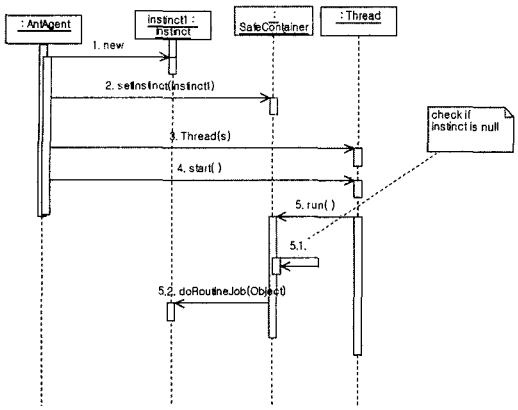


그림 4 Safe-Container 상호작용 다이어그램

결과: 하나의 Safe-Container는 Container를 올바르게 사용하는 하나의 방법을 정의한 것이다. 다른 방법을 정의한 Safe-Container2, Safe-Container3가 나중에 등장할 수 있다. 기본 기능을 갖춘 Container와 그 올바른 용법을 정의한 Safe-Container의 분리라는 점에서 고려사항의 분리의 실행 의의를 갖고 있다. 또한 개발자의 버그를 미연에 방지할 수 있다는 점에서 공격적 프로그래밍을 지원한다.

Credits: 이 패턴을 구현하기 위해 Template Method 디자인 패턴을 사용했다. 둘은 약간의 차이가 있다. Template Method는 정해진 알고리즘의 순서 내에서 하위 클래스가 각 단계를 재정의할 수 있는 능력에 초점을 맞추었다. 반면에 Safe-Container의 초점은 미리 버그가 발생하기 쉬운 작업의 순서를 미리 정의하고 Safe-Container 내에서 미리 자주 발생하는 오류의 검사를 하는 것이다.

바. Trigger Design Pattern

Controller는 모든 중요한 결정을 내리는 두뇌의 역할을 수행한다. 그러기 위해서는 Sensor와 Instinct들이 자신들의 작업을 수행 중에 Controller에게 보고할 수 있는 메커니즘이 있어야 한다. 신호등 관리 에이전트에서 다음 시나리오를 보자.

> Controller는 교통체증이 정상상태의 150%를 넘어선 경우에만 자신에게 알려달라고 Sensor에게 명령한다

이 시나리오에서 유추할 수 있는 Controller와 Sensor간의 상호작용은 2단계이다. Controller가 Sensor에게 자신을 호출해야 할 조건을 알려준다. Sensor는 계속 환경을 감지하다가, 그 조건이 만족되면 Controller의 특정 메소드를 호출한다. 물론 Controller의 자율성을 보장하기 위해 그 메소드의 수행은 비동기적으로 이루어지기 때문에, 실제 수행 시점은 Controller가 결정한다.

이름: Trigger

의도: Controller가 Sensor와 Instinct를 통제하고 통신할 수 있는 메커니즘을 정의한다.

구조:

클래스 이름: Trigger	
책무:	상호작용:
Controller에게 보고할 시점을 판단할 수 있는 로직을 정의한다. 해당 조건이 만족되었을 때 Controller에게 알려줄 자신의 ID를 알고 있다.	Controller

클래스 이름: Triggered	
책무:	상호작용:
트리거를 받아들여서 그들의 리스트를 관리한다. 변화가 있을 때마다 그 변화를 Controller에게 보고할지 Trigger에게 물어본다.	Trigger

상호작용:

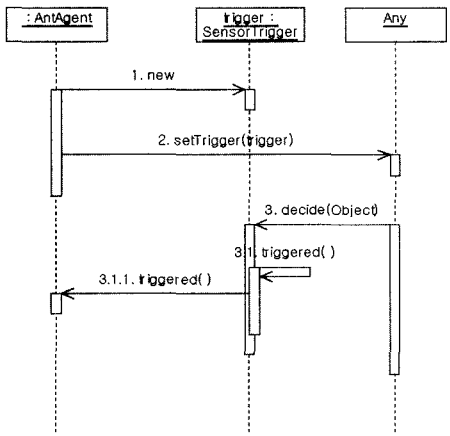


그림 5 Trigger 상호작용 다이어그램

결과: Controller와 Sensor, Instinct의 상호작용을 클래스로 식별해 그에 따른 책무를 Trigger에 캡슐화시켰다. 또한 CSA의 핵심인 Controller가 타 컴포넌트들과 상호작용하는 접점을 Trigger를 통해 제거시킴으로서 결합도를 감소시켰다.

Credits: 이름에서 유추할 수 있듯이 이 패턴의 아이디어는 데이타베이스의 Trigger에서 얻었다. 트리거에 해당하는 기능은 MS 윈도우에서 사용된 콜백(call-back), 자바에서 사용된 Observer 패턴 등 여러 방법으로 구현 가능하다. 본 패턴은 현재 알려진 방법 중 가장 널리 쓰이는 Observer 패턴을 적용하였다. 콜백이 아닌 Observer 패턴을 적용한 이유는 Controller와 타 컴포넌트간 결합도를 최소화시키기 위해서이다.

지금까지 각 패턴들을 하나씩 패턴 기술 형태에 맞추어서 기술하였다. 이번에는 그림 1에서 정의한 반응적

프레임워크 패턴 언어의 연관 관계와 순서에 대해 논한다.

설계의 첫 번째 단계는 아키텍처 스타일을 선택하는 것이다. 반응적 에이전트에서는 지금까지 CSA 모델이 가장 널리 사용되어 검증되었다. 두번째 아키텍처적 결정 사항은 자율성을 지원하기 위해 프로세스와 쓰레드 중 하나를 선택하는 것이다. 이 부분은 성능과 관계된 부분으로 문제 영역에 적절하게 선택하면 된다. 이 두 가지는 본 논문에서 새롭게 정의한 것이 아니다. 본 패턴 언어의 완성도를 위해 기존의 연구에서 차용한 것이다.

이 두 가지 아키텍처적 선택을 했으면 이를 지원하기 위한 세부 설계를 해야 한다. 이 때 적용 가능한 패턴은 Instinct 디자인 패턴과 Secretary 디자인 패턴이다. Secretary 패턴은 CSA의 세 컴포넌트가 서로 취급하는 정보의 수준이 다르기 때문에 이를 중계해주는 역할을 했다. Instinct는 CSA의 세 컴포넌트가 명확한 경계선을 만들고 Controller가 비대해지는 것을 막기 위해 적용되었다. 그렇지만 Instinct의 도입으로 인해 기존의 하부구조 서비스가 변경될 필요가 생겼다. 하지만 기존의 하부구조 서비스에 새로운 기능을 무작위로 추가하는 것은 하부구조 서비스의 모놀리식 구조화를 초래할 수 있기 때문에 Container 디자인 패턴이 도입되었다. Container의 도입으로 인해 Instinct의 생명주기에 대한 강력한 통제가 가능해졌지만 강력한 기능 제공으로 인해 새로운 버그 가능성이 존재하게 되었다. 이 새로운 고려사항을 분리시키기 위해 Safe-Container 디자인 패턴을 적용하였다.

본 프레임워크에서는 이들을 나열한 순서대로 적용하였다. 하지만 각 패턴은 서로 독립되어 있으므로 필요하다면 별도로 적용하는 것도 가능할 것이다. 예를 들어 Safe-Container의 종류가 한가지밖에 없을 경우 Container와 Safe-Container를 분리하지 않고 하나만 적용할 수도 있다.

4. 평 가

4.1 평가방법

ATAM[1](Architecture Trade-off Analysis Method)은 미국 카네기멜론 대학의 소프트웨어공학 연구소(SEI, Software Engineering Institute)에서 개발한 아키텍처 평가 방법이다. 시나리오에 기반하여 변경용이성(modifiability)을 평가하기 위한 방법인 SAAM(Software Architecture Analysis Method)을 발전시켜 여러 품질 속성을 평가하기 위한 방법으로, 본 논문에서는 제안한 패턴 언어와 해당 품질 속성과의 관계의 정확성을 검증하고, 패턴 언어의 정확한 사용처를 기술하기 위한 의도로 사용하였다[18].

본 패턴 언어가 추구하는 품질 속성은 성능, 안정성과 같이 평가 방법이 널리 알려진 품질이 아니라 변경용이성이라는 평가하기 힘든 속성이다. 따라서 이에 특화된 방법이 필요하다. ATAM의 시나리오는 항상 정량적인 판단을 내릴 수 있지는 않지만 정량적, 정성적으로 판단을 객관적으로 내릴 수 있는 수단을 제공한다. 또한 품질속성 유틸리티 트리는 각 시나리오들이 모여서 어떻게 상승 작용을 일으키는지 표현할 수 있는 정형적인 수단을 제공한다.

시나리오란 '이식성'과 같은 평가하기 애매한 품질을 윈도우 2000 Server 플랫폼에서 SUN-OS 5.6 플랫폼으로 변경했을 때 수정되어야 하는 컴포넌트의 개수는 몇 개인가'와 같은 시험 가능한 문장으로 정제한 것이다. 품질 속성 유틸리티 트리란 각 시나리오들이 향상시키는 품질을 각 시나리오의 상위 노드로 정의하고 그 각 노드들이 모여서 어떠한 품질들을 향상시키는지 트리 형태로 표현한 것을 말한다. 시나리오와 품질 속성 유틸리티 트리는 각 패턴과 그들의 합인 패턴 언어가 발현하는 품질 속성간의 관계를 정확하게 평가하기 위한 적절한 수단을 제공한다.

본 패턴 언어의 목적은 성능, 가용성과 같이 정량적으로 측정할 수 있는 품질의 획득이 아니다. 또한 프레임워크는 아직 애플리케이션에 비해 일반적이지 않기 때문에 구체적인 평가 기준이 표준으로서 제시된 바가 없다[2]. ATAM은 이러한 여건에서 특정 설계를 평가하기에 적합하다. ATAM은 특정 시스템을 평가하기 위해 만들어진 시나리오와 모두가 공감할 수 있는 일반적인 품질 속성 이 두가지를 연결하는 품질 속성 유틸리티 트리를 이용하기 때문에 프레임워크와 같이 일반적인 평가 기준이 없는 경우에도 매우 유용하게 사용할 수 있다.

이러한 목적에 맞게 본 논문의 ATAM 적용 결과물의 형태는 패턴 언어의 미세한 품질 속성들을 드러낼 수 있도록 각 패턴들을 평가할 수 있는 시나리오가 중심이 될 것이다. 그리고 이들이 전체로서 모였을 때 어떤 품질을 발현할 수 있는지 표현하기 위해 품질 속성 유틸리티 트리를 기술한다. 마지막으로 각 시나리오의 적법성을 나타내기 위한 시나리오 분석이 사용된다. 정리하면 본 패턴 언어의 평가를 위한 ATAM 결과물은 다음과 같다.

1. 품질 속성 유틸리티 트리: 최상위 노드에는 이 패턴 언어를 통해서 추구하고자 하는 하나의 목표를 잡는다. 최하위 노드에는 시나리오를 배치한다. 그리고 그 중간 노드에 각 시나리오가 확보하는 품질들을 배치한다. 이 때 하위 노드의 품질에 연결된 상위 노드의 품질은 하위 노드의 품질에 의해 상위 노드의 품질이 향상

됨을 의미한다. 본 논문에서는 보다 미세한 표현을 위해 상위노드의 품질 속성을 획득하기 위한 구체적 기술까지 확장함으로써 각 시나리오가 어떻게 품질 속성을 향상시켰는지 정확히 표현했다.

2. 시나리오: 각 시나리오는 영향요인, 아키텍처적 결정, 결과의 3개 노드로 구성된다. 여기서 아키텍처적 결정이 각각의 패턴에 해당한다.

3. 시나리오 분석: 각 시나리오가 적법한지 분석한 결과로 시나리오의 유형에 따라 정량적일 수도 있고 정성적일 수도 있다.

4.2 품질 속성 Utility Tree

유틸리티 트리는 1차적으로 단계 5에서 하향식 방법을 통해 연역적으로 만들어졌다. 여기서 분석된 결과를 토대로 단계 7에서 2차적으로 구성된다. 이 때는 상향식 방법을 통해 작성된다. 이렇게 2차례에 걸쳐 정제된 유틸리티 트리는 평가하고자 하는 아키텍처가 특정 품질 요소의 측면들을 얼마나 잘 확보하고 있는지 판단할 수 있는 기준이 된다.

본 연구에서 평가하고자 한 품질은 이 패턴 언어를 사용한 프레임워크가 얼마나 변경 용이성을 획득할 수 있느냐 하는 것이었다. 이를 객관적으로 정확히 평가하기 위한 방법의 표준은 사실상 존재하지 않는다. 그렇기 때문에 SEI에서 SAAM을 개발한 것이다[18].

ATAM을 거쳐서 최종적으로 완성된 품질 속성 유틸리티 트리는 그림 6과 같다. 먼저 s1~s6는 6개의 시나리오를 의미한다. 각각의 시나리오는 3.2.3절에서 소개한 패턴을 평가하기 위한 시나리오다. 각각의 시나리오는 다음 절에서 설명하고 여기에선 각 노드들의 의미와 관계에 대해 살펴본다.

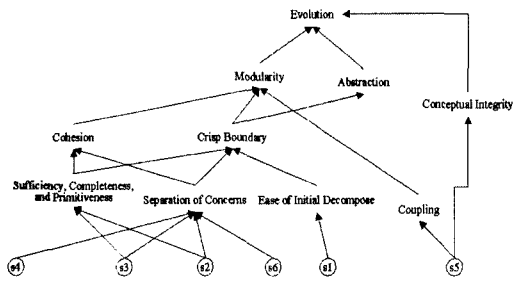


그림 6 반응적 에이전트 프레임워크 패턴 언어 품질 속성 유틸리티 트리

그림 6에서 상위 노드는 화살표로 연결된 하위 노드(들)에 의해 그 품질이 향상되었음을 의미한다. 굳이 이렇게 몇 단계를 거쳐서 시나리오들이 최종적으로 변경 용이성이라는 품질 속성의 향상을 표현한 것은 각 시나리오들의 품질을 보다 정확하고 미세하게 표현하기 위해서이다. 예를 들어 [s1 → 추상화 → 진화]로 품질을 표현한 경우와 [s1 → 진화]로 표현한 경우를 비교해보자. 후자의 경우는 s1이 어떻게 변경용이성에 도움을 주는지 논리적 비약이 존재한다. 하지만 전자는 그 중간 단계를 잘 표현해 준다. 따라서 그림 6과 같이 약간은 복잡한 유틸리티 트리는 각 시나리오로 인해 향상된 품질 속성을 보다 미세한 수준에서 표현할 수 있다.

4.3 시나리오

이름	시나리오 1(s1)
영향 요인	반응적 에이전트의 행위를 유스케이스 모델링한다. 각 유스케이스별로 에이전트에게 할당되어야 할 문구를 모두 추출한다. 모든 문구를 CSA 중의 한 컴포넌트로 할당한다.
아키텍처적 결정	CSA
결과	모든 문구가 명확한 경계선을 이루면서 세 개의 컴포넌트에 할당되었는가?

이름	시나리오 2(s2)
영향 요인	반응적 에이전트의 행위를 유스케이스 모델링한다. 각 유스케이스별로 에이전트에게 할당되어야 할 문구를 모두 추출한다. 모든 문구를 CSA + Instinct 중의 한 컴포넌트로 할당한다.
아키텍처적 결정	Instinct 디자인 패턴
결과	4개의 컴포넌트에 할당된 문구들은 명확히 서로 독립된 고려사항들을 가지고 있는가? 각 컴포넌트는 한 명이 독립적으로 구현하기에 충분히 작은 크기인가?

이름	시나리오3(s3)
영향 요인	시나리오2에서 Instinct에게 할당된 문구들 중 비즈니스 로직은 Instinct에 남겨두고, 하부구조 서비스에 해당하는 내용만을 추출해서 Container에 할당한다.
아키텍처적 결정	Container 디자인 패턴
결과	4개의 컴포넌트에 할당된 문구들은 명확히 서로 독립된 고려사항들을 가지고 있는가? 각 컴포넌트는 한 명이 독립적으로 구현하기에 충분히 작은 크기인가?

이름	시나리오4(s4)
영향 요인	Container의 미세한 기능들을 적법한 절차에 의해서 이용 가능한 패키지로 묶는다.
아키텍처적 결정	Safe-Container 디자인 패턴
결과	컨테이너를 직접 사용했을 경우 error-prone한 코드가 나올 가능성이 있는가? 그렇게 해서 만들어진 각각의 묶음이 여러 개 존재하는가? 그들은 실행시간에 교체될 필요가 있는 것들인가? 그 각각의 묶음은 명확히 구분되는 서로 다른 고려사항을 가지고 있는가?

이름	시나리오5(s5)
영향 요인	Controller의 함수를 호출해야 하는 객체의 개수
아키텍처적 결정	Trigger 디자인 패턴
결과	Controller와의 상호 작용 방법의 일원화로 개념적 통일성이 증가하였는가? 상호 의존 관계가 제거되어서 Controller와 타 객체들간의 결합력이 감소하였는가?

이름	시나리오6(s6)
영향 요인	Controller와 Sensor, Actuator가 다루는 정보의 추상화 수준 차이
아키텍처적 결정	Secretary 디자인 패턴
결과	추상화 수준이 얼마나 차이가 나는가? 서로 다른 추상화 수준의 정보를 중계하기 위한 컴포넌트인 Secretary가 도입됨으로 인해서 정보 수준의 조정이라는 책임이 Secretary로 완전히 이동되었는가?

4.4 시나리오 분석

앞에서 제시한 각 시나리오를 분석하겠다. 먼저 영향 요인을 제시하고 패턴을 해결책으로 적용한 결과가 앞에서 제시한 결과 항목에 얼마나 부합하는지 살펴본다.

가. 시나리오 1(s1)

에이전트는 객체보다 상대적으로 추상화 수준이 높다[23]. 한번에 전체를 생각하기에는 복잡도가 높기 때문에 초기에 최대한 독립적인 경계선을 가지고 적절한 크기로의 분할이 필요하다. 그럼에도 불구하고 각각의 분할된 조각이 서로 많이 연관되어 서로 밀접하게 연동되는 경우 각각의 분할된 조각을 맡은 개발자가 독립적으로 작업할 수 없다. 또한 각각의 분할 된 컴포넌트가 적절한 수준에서 정보를 은폐했을 때 전체 컴포넌트의 변경 용이성에 대한 가능성이 보다 높아진다. 개발자가 감당할 수 있는 크기로의 분할은 이후에 적절한 사고를 하는데 도움을 준다.

본 연구에서 예제로 사용한 신호등 관리 에이전트 시스템을 유스케이스로 모델링한 결과는 다음과 같다. 다음 유스케이스 다이어그램에서 일부분 혹은 전체를 에이전트가 수행해야 할 유스케이스는 시작, 끝, 초기 정책 설정, 교통 상황 모니터링, 차량 통제의 5개이다.

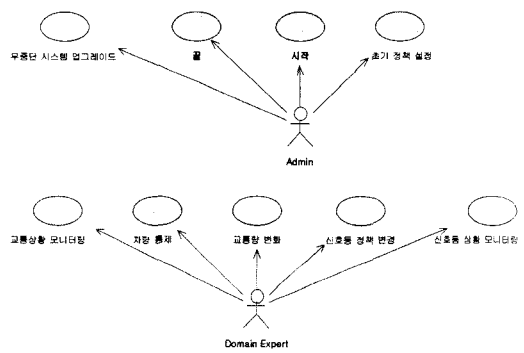


그림 7 신호등 관리 에이전트 유스케이스 다이어그램

5개의 유스케이스의 기능들을 시나리오 1의 해결책인 CSA의 3개 컴포넌트에 각각 할당하면 다음과 같다.

C(Controller, Controller)

1. 시작 유스케이스
   - ■ 에이전트의 기능들을 초기화 하고 Sensor와 Actuator를 구동한다.
2. 끝 유스케이스
   - ■ 에이전트의 상태를 저장하고 종료한다.
3. 초기 정책 설정 유스케이스
   - ■ 관리자가 설정한 정책을 입력 받아 에이전트의 이후 계획을 초기화하고 계획한다.
4. 교통 상황 모니터링 유스케이스
   - ■ N/A
5. 차량 통제 유스케이스
   - ■ 교통량이 임계치를 넘어설 경우 에이전트가 개입해야 할 상황인지 판단한다.
   - ■ 에이전트가 개입하기로 판단했으면, 신호등의 점등 주기를 적절하게 변경한다.

S(Sensor, Sensor)

1. 시작 유스케이스
   - ■ Controller가 설정한 초기화 정책대로 구동한다.
2. 끝 유스케이스
   - ■ N/A
3. 초기 정책 설정 유스케이스
   - ■ Controller가 설정한 정책으로 자신을 초기화한다.
4. 교통 상황 모니터링 유스케이스
   - ■ 지속적으로 교통 상황을 감시한다.
   - ■ Controller가 설정한 임계치를 넘어서면 Controller가 지정한 행위를 한다.
5. 차량 통제 유스케이스
   - ■ N/A

A(Actuator, Actuator)

1. 시작 유스케이스
   - ■ N/A
2. 끝 유스케이스
   - ■ N/A
3. 초기 정책 설정 유스케이스
   - ■ N/A
4. 교통 상황 모니터링 유스케이스
   - ■ N/A
5. 차량 통제 유스케이스
   - ■ Controller가 지정한대로 각 신호등의 점등 주기를 변경한다.

CSA 세 컴포넌트에 각각 할당된 기능들은 서로간의 경계가 명확하다. 직관적으로 봐서 어느 한 쪽에 기술된 기능이 다른 컴포넌트로 할당되야 하는 것이 아닌가 하는 애매모호성이 없다.

실제로 이 방법을 적용한 결과 분석 단계에서의 기능들은 모두 CSA의 세 컴포넌트에 명확한 경계선을 구성하면서 분배되었다. 하지만 실제로 그들간의 상호 작용이나 정보 교환 방법 등 설계 상의 기능은 특정 컴포넌트에 배분하기 애매모호한 경우가 많았다. 또한 Controller에 할당된 문구가 상대적으로 많아 Controller가 비대함을 알 수 있다. 이후에 등장한 s3~s6은 CSA 모델의 그러한 문제들을 하나씩 해결한 패턴들이다.

나. 시나리오2(s2)

CSA로 에이전트의 기능을 분할한 결과 Controller가 가장 비대한 컴포넌트가 되었다. 이는 에이전트의 특성상 당연한 결과이지만, 에이전트의 지능이 높아질수록 이런 현상은 가속화 되어 모놀리식해질 우려가 있다. Controller의 내부 구조를 한 번 더 정제할 필요가 있다.

신호등 관리 에이전트 중 Controller와 Instinct의 기능을 비교하면 다음과 같다. Sensor와 Actuator의 경우는 시나리오1과 동일하므로 생략한다.

Controller:

→ 에이전트의 기능들을 초기화하고 Sensor와 Actuator를 구동한다.

→ 에이전트의 상태를 저장하고 종료한다.

→ 관리자가 설정한 정책을 입력 받아 에이전트의 이후 계획을 초기화하고 계획한다.

→ 교통량이 임계치를 넘어설 경우 에이전트가 개입해야 할 상황인지 판단한다.

→ 에이전트가 개입하기로 판단했으면, 신호등의 점등 주기를 적절하게 변경한다.

Instinct:

→ 교통량의 변화가 미리 정의된 임계치를 넘지 않으면, 정해진 시간 간격으로 신호를 바꾼다.

→ 주기적으로 자신이 관리하는 지역의 교통량을 인접한 에이전트에게 알려준다.

Controller와 Instinct의 차이는 명확하다. 지능적으로 판단을 내려야 하고 그 판단의 입력이 Sensor, 출력이 Actuator면 Controller이다. 그렇지 않고 지속적으로 동일한 동작을 반복하는 것이 목적이라면 Instinct이다. 즉 Controller의 고려사항이 이성적인 판단을 내리는 지능이라면, Instinct의 고려사항은 본능이다. 이 2개의 컴포넌트는 서로 다른 서로 다른 고려사항을 명확히 분리해 준다. 또한 Controller에 뭉쳐져 있던 2개의 기능이 2개의 컴포넌트로 분리되었으므로 그 크기가 작아졌다.

다. 시나리오3(s3)

Instinct는 그 특성상 Controller와는 생명주기와 동작 형태도 다르다. 비즈니스 로직은 둘 다 애플리케이션마다 다르니까 재사용 가능한 부분을 추출하는 것이 어렵지만 하부구조 서비스는 그렇지 않다. 예를 들어 스레드의 생성과 소멸 혹은 그 스레드와의 통신 방식 등은 대부분의 애플리케이션이 동일하다. 하지만 Controller와 Instinct는 하부구조 서비스의 행태가 서로 다르기 때문에 분리하는 것이 Instinct가 Controller의 하부구조 서비스를 그대로 이용하는 것은 바람직하지 않다. 사용하는 행태가 다르다는 것은 하부구조 서비스 동작 행태가 다름을 의미하기 때문이다.

그 둘이 얼마나 다른지 파악하기 위해서는 그 둘의 사용 시나리오를 비교해서 차이점을 분석하면 알 수 있다. 시나리오 별 차이점을 살펴 보면 다음과 같다.

1. Controller는 Sensor로부터 정보를 받아 자신의 지능으로 판단을 내려 Actuator를 통해 명령을 내린다. 반면 Instinct는 초기에 정해진 규칙대로만 행동한다.
2. Controller의 생명주기는 소속 에이전트와 동일하다. 반면 Instinct는 소속 에이전트보다 늦게 생성되어서 먼저 소멸한다.
3. Controller는 타 에이전트와 상호작용을 한다. 반면 Instinct는 소속 에이전트 내부의 컴포넌트들과만 상호작용을 하고 외부와는 단절되어 있다.
4. Controller에 지능을 부여할 경우 JESS와 같은 외부 프레임워크와 결합될 수 있다. 에이전트의 특성상 이는 매우 빈번할 것으로 예상된다. 반면 Instinct는 그렇지 않다.

시나리오에서 보듯이 Controller와 Instinct의 동작 행태는 매우 다르다. 따라서 이 둘의 하부구조 서비스는 그 지원기능과 범위가 다를 수 밖에 없다. 이는 고려 사항의 분리가 적용되어야 함을 의미한다. 또한 이렇게 나누어진 컴포넌트는 그 크기가 작아졌으므로 원시성이 향상되었다.

라. 시나리오4(s4)

Container가 계속적인 진화를 거칠 경우 그 기능이 점점 많아져서 지나치게 다양한 조합의 사용 사례가 만들어 질 수 있다. 이런 미세한 기능들의 증가는 컨테이너 사용의 강력함을 증가시키지만 그 반대급부로서 기능들을 잘못 조합해서 사용하는 것이 가능해진다. 앞에서 예시한 Null Pointer Exception 같은 경우이다.

본 프레임워크는 버전이 낮아서 아직은 그러한 다양한 사용사례가 발견되지 않았다. 따라서 시나리오 4에서는 SAAM과 같이 가상의 미래의 변화를 상정하여 그러한 사용사례의 존재 가능성을 타진해 본다. 먼저 Container에서 제공하는 원시 기능을 나열하고, 이들을 조합해서 잘못 사용할 수 있는 사례를 보자.

1. 원시기능 : Instinct 추가/삭제/시작/중지/정지/재시작
2. 잘못 사용할 수 있는 사례 : 추가 없이 삭제, 삭제된 Instinct를 이용한 조작(시작/중지/정지/재시작)

Container는 int, char과 같은 원시기능을 제공한다. 반면 Safe-Container는 struct와 같은 복합기능을 제공한다. 프로그래밍 언어에서 검증되었듯이 이 2개의 고려사항의 분리는 많은 효과가 있다.

마. 시나리오5(s5)

CSA중 집적도와 중요성이 가장 높은 Controller는 프레임워크의 진화에 가장 큰 걸림돌이다. 왜냐하면 Controller에 영향을 주는 외부의 변화 요인이 Sensor와 Actuator에 비해 상대적으로 많고, 높은 집적도 때문에 수정이 어렵기 때문이다. 이 문제의 해결책은 타 컴포넌트와 Controller 간의 결합력을 감소시켜서 Controller의 고립도를 높이는 것이다. 이번 시나리오에서는 Trigger 패턴의 도입으로 Controller의 결합력이 얼마나 감소되었는지를 판단한다.

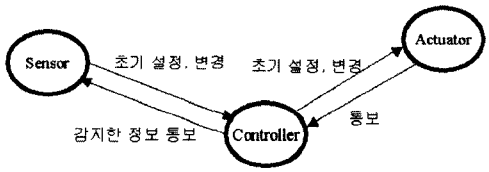


그림 8 Trigger 디자인 패턴 적용 이전 관계

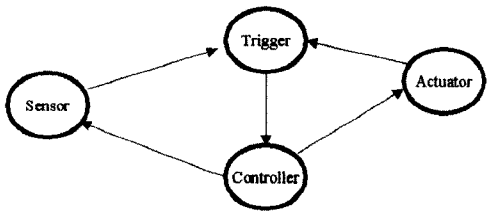


그림 9 Trigger 디자인 패턴 적용 이후 관계

Trigger 디자인 패턴을 적용한 경우를 살펴보면 상호참조가 제거되었다. 또한 Controller의 변화에 의해서 영향을 받는 컴포넌트가 Trigger 하나로 감소하였다. 실제로 구현할 때에는 Trigger의 구현은 Controller와 같이 존재하므로 같이 변화한다. 변화 시점이 같으므로 이 둘 간의 높은 결합력은 문제되지 않는다. 그리고 Controller와 통신 방법이 Trigger라는 한가지 방법으로 제한되었으므로 개념적 통일성이 증가했다.

바. 시나리오6(s6)

시나리오 6는 Controller와 Sensor/Actuator 간에 서로 취급하는 정보의 추상화 수준이 얼마나 다른지 비교해보면 그 효과를 판별할 수 있다. 교통량이 급증해서 신호등의 점등 주기를 변경하는 경우 일어나는 상황을 예로 해서 보자.

표 2 CSA 컴포넌트들이 취급하는 정보

정보의 종류	정보의 내용
Sensor가 감지한 정보	북쪽 신호등: 현재 좌회전 대기 차량이 18대, 직진 대기 차량 30대
Controller가 Sensor로부터 기대하는 정보	북쪽 신호등: 평상시보다 좌회전 대기 차량 10% 증가, 직진 대기 차량 35% 증가
Actuator가 Controller로부터 기대하는 정보	북쪽 신호등: 직진 신호를 현재 1분에서 1분 10초로 증가시킬 것

표 2에서 볼 수 있듯이 Controller가 사용하는 정보는 지능적인 판단을 내리는데 필요한 가공된 정보이다. 반면에 Sensor와 Actuator가 사용하는 정보는 환경을 인식하기 위한 가공 전의 정보이다. 이 둘 간의 차이 때문에 Secretary 디자인 패턴이 필요한 것이다.

5. 결 론

점점 확대되는 에이전트의 사용을 뒷받침 하기 위해선 에이전트의 하부구조의 핵심인 에이전트 프레임워크의 발전이 필수적이다. 현재 가장 널리 사용되는 객체지향에 비해 상대적으로 적용 경험이 짧은 에이전트 분야에서는 서로의 경험을 공유해서 간접 경험의 양과 질을 높이는 것이 필요하다. 이를 위해 본 논문에서는 패턴 언어를 통한 방법을 제시했다.

패턴 언어는 각각의 패턴들이 모여서 하나의 공통 선을 창출해내야 한다. 본 논문에서는 반응적 에이전트 아키텍처의 진화를 테마로 해서 총 7개의 패턴을 제시했다. 각각의 패턴은 서로 유기적으로 연결되어서 반응적 프레임워크 패턴언어를 구성하였다.

각각의 패턴들은 모여서 반응적 에이전트 프레임워크 패턴 언어를 구성하고 이들 각각의 패턴이 향상시킨 품질 속성은 최종적으로 애플리케이션과 프레임워크의 변경용이성을 지원하는 품질 속성을 상승시켰다.

본 연구를 통해서 에이전트 프레임워크 개발자들이 서로의 경험을 오픈 소스뿐만 아니라 패턴과 패턴 언어를 통해서 공유할 수 있기를 기대한다. 패턴을 통한 경험의 공유는 개발자들의 간접 경험을 증가시켜 시행착오를 줄이고 그들의 아키텍처를 보다 견고히 해줄 것이다. 구현 하부구조의 성숙은 최종적으로 에이전트 지향 소프트웨어 공학의 발전으로 이어질 것으로 기대된다.

그러나 본 논문에서 제시한 문제 영역은 반응적 에이전트 아키텍처라는 작은 영역이다. 팽창하는 에이전트 영역을 본격적으로 뒷받침 하기 위해서는 각 에이전트 분야별로 다양한 패턴 언어가 발표되서 전체적인 성숙도가 높아질 필요가 있다. 과거 객체지향과 CBD의 성

장을 볼 때 기술과 경험이 축적을 거쳐 돌파로 전환한 시점은 실제로 그 패러다임을 구현할 수 있는 하부 기술이 대중적으로 퍼지기 시작했을 때였다.

향후 이 연구를 보다 발전시켜 반응적 에이전트뿐 아니라 다른 여러 종류의 에이전트에도 적용할 수 있는 일반적인 패턴 언어가 등장할 것으로 기대된다. 그러나 이는 추측에 의해서 만들어질 수 있는 것이 아닌 만큼, 각 분야별로 특화된 패턴 언어가 먼저 발표되고 이들을 일반화시켜야 올바른 패턴 언어가 발견될 것이다.

참 고 문 헌

[1] Paul Clements, Rick Kazman, Mark Klen. Evaluating Software Architecture. Addison Wesley. 2002.

[2] Barbara Gengler. Silicon super-agents, AustralianIT, April 30 2002, see http://australianit.news.com.au/articles/0,7204,4210558%5E15397%5E%5Enbv%5E,00.html

[3] Mohamed E. Fayad. A Framework for Agent Systems. Implementing Application Frameworks, Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editor. Willey, 1999.

[4] The Foundation for Intelligent Physical Agents. See http://www.fipa.org

[5] OMG Agent Platform Special Interest Group. Agent Technology Green Paper Ver. 1.0. See http://www.objs.com/agent/index.html

[6] FIPA-OS. http://fipa-os.sourceforge.net/

[7] JADE, http://sharon.cselt.it/projects/jade/

[8] AGLET, http://www.trl.ibm.co.jp/aglets/

[9] Kent Beck and Ralph Johnson. Patterns generate architectures. In Proceedings ECOOP'94, volume 821 of LNCS, pages 139--149. Springer-Verlag, July 1994.

[11] Elizabeth A. Kendall, P.V. Murali Krishna, Chirag V. Pathak, and C.B. Suresh. A Framework for Agent Systems. In Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editors, Implementing Application Frameworks: Object-Oriented Frameworks at Work, John Wiley & Sons, 1999.

[12] http://fipa-os.sourceforge.net/docs/presentations/fipa-and-fipaos.pdf 8Page.

[13] H. S. Nwana and D. T. Ndumu. A Brief Introduction to Software Agent Technology. In Nicholas R. Jennings and Michael J. Wooldridge, editors, Agent Technology: Foundations, Applications, and Markets, Springer-Verlag, 1998.

[14] Onn Shehory. Architectural Properties of Multi-Agent Systems, tech. report CMU-RI-TR-98-28, Robotics Institute, Carnegie Mellon University, December, 1998.

[15] Parnas, D.L., "Software Aging" in Proceedings of the 16th International Conference on Software Engineering", Sorento Italy, IEEE Press, 279-287, May 16-21/94.

[16] Scott W. Ambler. Process Patterns. Building Large-Scale Systems Using Object-Oriented Technology. Cambridge University Press, 1998.

[17] Martin Fowler and Kendal Scott. UML Distilled 2nd edition. Addison-Wesley. 1999.

[18] Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice. Addison-Wesley. 1998.

[19] Don Roberts and Ralph Johnson. Patterns for Evolving Frameworks. Pattern Languages of Program Design 3. Robert Martin, Dirk Riehle, and Frank Buschmann, editor. Addison Wesley, 1998.

[20] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000D. [26]Kinny, M. George, and A. Rao. A methodology and modelling technique for systems of BDI agents. MAA-MAW'96, LNAI Volume 1038, Springer-Verlag, 1996, pages56-71.

[21] Kinny, M., George and A. Rao, A methodology and Modelling technque for Systems of BDI agents, MAA-KAK'96 LNAEI Volume 1938, Springer-Verlay, 1996, Page 56-71.

[22] Mary Shaw and David Garlan. Software Architecture: Perspectives on and Emerging Discipline. Prentice Hall, 1996.

[23] Alex L.G. Hayzelden and John Bigham. Software Agents for Future Communication. Springer, 1998.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. Proceedings European Conference on Object-Oriented Programming. 1997.

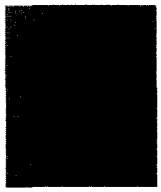
[25] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed, "Autopilot: Adaptive Control of Distributed Applications," Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, July 1998.

[26] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[27] Meyer, B. Applying "Design by Contract," Computer 25(10), October 1992, 40-51.

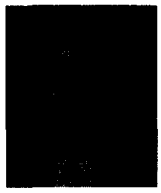
박 성 운

2000년 2월 서강대학교 컴퓨터학과(공학학사). 2003년 2월 서강대학교 컴퓨터학과(공학석사). 現 아이티플러스. 관심분야는 소프트웨어 아키텍처, 패턴



정 재 민

1999년 2월 서강대학교 컴퓨터학과(공학학사). 現 서강대학교 컴퓨터학과 재학(공학석사). 관심분야는 소프트웨어 아키텍처, 패턴



박 수 용

1986년 2월 서강 대학교 전자계산학과(공학학사). 1988년 8월 Florida State University, Computer Science (M.S.) 1995년 5월 George Mason University, Information Technology (Ph.D.) 現 서강 대학교 컴퓨터학과 교수. 관심분야는 요구공학, 소프트웨어아키텍처, 에이전트 지향의 소프트웨어 공학