디자인 패턴을 이용한 Retargetable 시뮬레이터의 개발

김 영걸, 김 탁곤

한국과학기술원, 전기 및 전자공학과
시스템 모델링 시뮬레이션 연구실
URL: http://smsl.kaist.ac.kr
E-mail: {ygkim@smslab, tkim@ee}.kaist.ac.kr

Abstract

디자인 패턴은 소프트웨어 -특히, 객체지향 소프트웨어- 의 개발시 재 사용성을 높기 위해서 사용되며, 이는 상속(Inheritance)과 같은 코드레벨 재사용 (code reuse)보다 높은 레벨의 디자인 재사용 (design reuse)을 가능하게 한다. 디자인 패턴은 구체적인 문제에 대해 구체적인 해를 제공하는 cookbook과는 달리, 추상적인 문제에 대해 추상적인 해를 제시함으로써, 비슷한 부류의 문제에 적용할 수 있으므로 높은 재 사용성을 보장한다. 본 논문은 Retargetable한 특성을 갖는 Instruction set simulator의 개발에 디자인 패턴을 적용한 예를 보여줌으로써, 재 사용성 및 확장성을 높이는 방안을 소개한다.

1. 서론

프로세서의 복잡도가 점차 증가하고 사용자의 요구사항이 다양해짐에 따라, 프로세서의 개발 소요 시간을 줄이는 것이 매우 중요한 연구 과제가 되고 있다 ([1]).

이러한 연구중의 하나로서, 프로세서의 아키텍처가 변함에 따라 그에 맞게 시뮬레이터와 컴파일러가 변하게 하는 연구([2],[3])가 진행되고 있으며, 이들을 각각 retargetable simulator 및 retargetable compiler라고 부른다. 본 논문은 retargetable simulator중에서도 명령어 집합을 시뮬레이션 할 수 있는 retargetable ISS를 개발함에 있어서 재 사용성 및 확장성을 높이기 위해서 디자인 패턴([4])을 적용한 것을 보여준다. 2장에서 Retargetable ISS의 개념과 구조를 간략히 소개하고, 3장에서 디자인 패턴에 대해 언급한 뒤, 4장에서는 Retargetable ISS의 개발에 적용된 디자인 패턴에 대해서 자세히 기술하겠다. 마지막으로 5장에서 결론을 맺는다.

2. Retargetable ISS

Retagetable Instruction Set Simulator - 앞으로 RISS라고 하겠다 - 는 프로세서의 명령어 집합의 구조가 변하여도 그에 맞게 시뮬레이션을 행하여 주는 시뮬레이터를 일컫는다. 기존의 방법으로는 아키텍처가 변하면 그에 맞게 시뮬레이터를 재 구현해야 하므로, 전체 개발 시간이 증가하게 되고 재작성으로 인한 에러의 유발 가능성이 높다. RISS를 이용하면 아키텍처가 허용되는 범위내에서 변할 때, 그에 맞게 시뮬레이터가 자동으로 생성 되므로 위에서 말한 단점을 극

복할 수 있다.

이렇게 좋은 장점이 있음에도 불구하고, RISS를 이용하기 위해서는 극복해야 할 문제점들이 몇 가지가 있게 된다. 첫 째, 자동화된 환경을 만들기 위하여 명령어 집합이 정확한 의미론에 의해서 만들어진 언어 - 주로 Architecture Description Language (ADL) 라고 한다 - 로 표현이 되어야 한다. 둘 째, 아키텍처 기술언어가 잘 정의 되었다 하더라도 이 정보를 RISS에게 넘겨주기 위한 방법론이 필요하게 된다. 세 째, Retargetability를 제공하는 명령어 집합 시뮬레이터를 구현하는 것 자체에도 많은 어려움이 따르게 된다.

2.1 아키텍처 기술언어 : READ

앞에서도 이미 밝혔듯이, retargetable한 특성을 가지는 simulator를 개발하기 위해서는 우선 아키텍처의 정보를 정확히 기술해 줄수 있는 언어가 먼저 정의되어야 한다. 본 연구에서는 아키텍처를 효율적으로 쉽게 기술하면서도 재사용성을 극대화 할 수 있도록 고안된 언어를 개발하였으며, 그 이름을 READ (Reusable Architecture Description Language)라고 명명하였다.

READ는 명령어 집합의 정보를 기술할 수 있도록 하기 위하여, 명령어들과 가능한 어드레싱 모드들 및 레지스터 파일등의 storage 정보를 기술하도록 정의 되어 있다. 또한, 성능 측정을 위해서 명령어별로 소요되는 싸이클의 수를 기술하도록 하였다. RISS는 이렇게 READ로 기술된 아키텍처의 정보를 받아서 자동으로 그에 맞는 시뮬레이터를 생성해 주게 된다.

2.2 RISS의 구조

RISS는 아키텍처를 명령어 집합 레벨에서 시뮬레이션 하기 위한 retargetable 시뮬레이터로서, generic한 시뮬레이션 엔진과 모델 합성기로 크게 나뉘어 진다. 모델 합성기는 READ로 기술된 아키텍처 정보로부터 시뮬레이션 엔진이 원하는 포맷에 맞게 모델을 자동으로 생성해 주는 역할을 하게 되며, 크게 decoding table 합성기와 model structure 합성기로 다시 나뉜다.

시뮬레이션 엔진은 generic하면서도 확장 가능하게 고안이 되었으며, 이를 위하여 template method 및 strategy pattern이 복합적으로 적용되었다. 이는 3장에서 설명하겠다. 기본적인 시뮬레이션 엔진의 핵심 부분은 *CodeManager*와 *Fetcher, Decoder* 및 *Executer* 그리고 *Scheduler*의 클래스로 이루어져 있다. 시뮬레이션의 목적이 성능 평가가 아니라 동작 검증만을 위해서라면 시뮬레이션의 효율을 위해서 Scheduler는 빠지게 된다.

3. 디자인 패턴

3.1 디자인 패턴의 필요성

객체 지향 소프트웨어를 개발하는 것은 상당히 힘든 작업이며, 재사용 가능한 객체 지향 소프트웨어를 개발하는 것은 더욱 힘들다고 할 수 있다. 우선 문제를 풀기 위한 적당한 객체들을 찾아야 하고, 그들을 적합한 granularity를 가지도록 클래스로 분류함과 동시에 상속 관계 (Inheritance hierarchy)를 정의해야 한다. 또한 당면한 문제에 맞도록 디자인함과 동시에 나중에 발생할 수 있는 문제나 요구사항을 잘 받아들일 수 있도록 설계가 되어야 하는 어려움이 있다. 객체 지향 소프트웨어의 초보 개발자들은 이러한 복잡함에 잘 대처하지 못하는 반면, 경험이 많은 전문가들은 실제로 좋은 설계 및 개발을 한다.

경험이 많은 전문가들은 모든 문제를 처음부터 끝까지 일일이 풀려고 하지 않으며, 과거에 잘 풀었던 해결책을 재사용 하려고 한다. 즉, 패턴을 잘 활용하면 많은 디자인상의 결정들이 자동으로 되는 것이다. 간단히 말해서, 디자인 패턴은 개발자가 보다 빨리 제

대로 된 디자인을 할 수 있도록 도와주는 역할을 한다.

3.2 디자인 패턴의 정의

패턴이라는 용어는 건축가였던 Alexander에 의해서 만들어졌는데, 그는 패턴에 대해서 다음과 같이 말하였다.

"각각의 패턴은 우리가 사는 환경에서 계속 반복되는 문제와 그 문제에 대한 해결책을 제시하며, 이를 계속 이용해서 똑같은 문제에 대해서 다시 처음부터 해결책을 찾는 일을 하지 않도록 하여 준다"

이 말은 건축 (건물)의 패턴에 대해서 언급한 것이지만, 소프트웨어의 설계에도 똑같이 적용되며, 특히 객체지향 소프트웨어에서 언급되는 디자인 패턴은 객체들과 그들간의 관계를 이용하여 문제와 해결책을 기술하게 된다. 일반적으로, 패턴은 다음과 같은 4가지의 필수적인 요소를 포함한다.

1. 패턴 이름 (pattern name)

패턴의 이름은 문제/해결책/문제가 일어난 상황을 기술하고, 엔지니어끼리 커뮤니케이션을 원할하게 하기 위해서 필요한 일종의 핸들이며, 프로그래밍 보다 높은 추상화 레벨에서 문제를 생각하고 토의할 수 있도록 하여 준다.

2. 문제 (problem)

문제는 언제 패턴을 적용해야 할지를 기술하여 주며, 문제자체에 대한 기술뿐만 아니라 그것이 발생한 상황을 설명하여 준다.

3. 해결책 (solution)

여기서 말하는 해결책은 구체적인 디자인이나 구현을 기술하는 것을 말하는 것은 아니다. 그 이유는 패턴 자체의 특징 및 장점이 많은 비슷한 상황에 적용할 수 있는 일종의 틀의 역할만 하기 때문이다. 따라서, 구체적인 문제에 대한 구체적인 해결책이 아니라 보다 추상적인 문제에 대한 추상적인 해결책을 기술하여 준다고 보면 된다. 특히, 객체지향의 디자인 패턴은 성공적인 디자인을 이루는 요소들, 그 요소들간의 관계, 책임 분배 및 협동 관계를 주로 기술하게 된다.

4. 결과 (consequences)

결과라는 것은 패턴을 적용하였을 때 일어나는 result와 trade-off를 의미하며, design상의 대안(alternative)들중 하나를 선택하거나 패턴을 적용하였을 때의 비용(cost)을 예상할 수 있다는 측면에서 중요하다고 할 수 있다.

4.Design Pattern을 이용한 RISS의 개발

RISS는 실제로 C++ 언어를 이용하여 구현이 되었으며, C++가 제공하는 객체지향성을 사용하여 디자인 패턴을 비교적 쉽게 적용할 수 있었다. 표 1을 보면 RISS를 개발하기 위해서 쓰인 디자인 패턴과 각각의 패턴이 어떤 개체의 디자인에 어떤 식으로 적용 되어있는지 나와있다. 이 중에서 특히 strategy pattern에 대해서 구체적으로 설명하고, 이 패턴이 RISS에서 어떻게 적용되었는지 설명하겠다.

표 1 RISS에 적용된 디자인 패턴

Factory Method	To create different kinds of registers	IntegerRegister & FloatingPointRegister
Builder	To build complex & hierarchical simulator template structure	SimConstructionDirector SimBuilder SimEngineBuilder
Bridge	To separate register abstraction & number representation (implementation)	IntegerRegister & RInteger
Facade	To manage a family of related commands to be sent to each class otherwise	SynthesizerManager SESManager
Template Method	To use same simulation step for simulation	SimCore::run()
Visitor	To visit only interested entities in PES for simulator synthesis	Synthesizer : public SESVisitor
Strategy	To use different policy for algorithms with same concept (Object polymorphism)	Executer::run() -> cur_instr->run()

strategy 패턴은 관련된 알고리즘들의 집합을 정의하고 각각을 encapsulate한 뒤, 서로 교환 가능하게 하여, 이것을 쓰는 client 객체와 상관없이 알고리즘이 변할 수 있도록 하여 주는 패턴으로서 Policy 패턴으로도 알려져 있다.

즉, 그림 1을 참조하면 *Strategy* 클래스는 알고리즘의 인터페이스를 제공하는 abstract 클래스이고, client 객체인 *Context*는 이 인터페이스를 이용하여 service를 request한다. 구체적인 알고리즘은 *Strategy* 클래스의 하위 클래스들인 *ConcreteStrategy* 클래스들에 의해서 정의된다. 여기서 중요한 점을 살펴 보자.

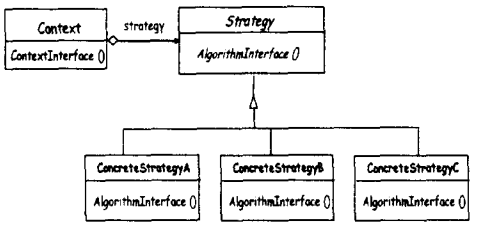

그림 1 Strategy 패턴

그림 1에서는 구체적인 strategy로서 3개의 concrete strategy를 보였지만, 만약 나중에 하나의 구체적인 strategy를 추가하고 싶으면 *Strategy* 클래스로부터 상속을 받고 해당하는 algorithm interface를 구현하면 된다. 이 때, client 객체인 *Context*의 코드는 전혀 변할 것이 없는데, 그 이유는 객체 지향 패러다임에서 지원되는 polymorphism에 의해서 자동으로 나중에 만든 클래스가 인식이 되기 때문이다. (dynamic binding)

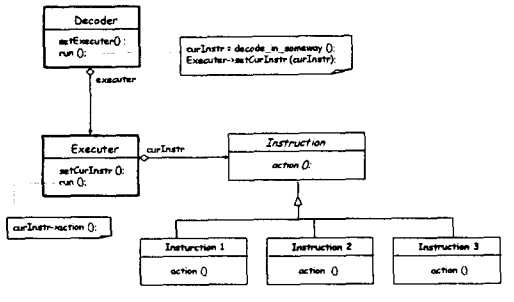

그림 2 Strategy 패턴의 적용 예

그림 2는 strategy 패턴을 RISS에 적용한 예를 보여 주고 있다. 그림에서 알 수 있듯이, Executer는 현재 수행해야 할 명령을 Decoder로부터 받아 와서, 해당 명령어의 run()이라는 method를 부르게 된다.

여기서 *Instruction* 클래스는 action()이라는 interface를 제공하고 실제의 action은 그 하위 클래스들에 의해서 구체적으로 정해진다. 즉, *Executer* 클래스는 그림 1의 패턴에서 *Context* 클래스에 해당하며, *Instruction* 클래스는 Strategy라는 abstract class에 해당한다. 하위 클래스들은 타겟 아키텍처의 명령어 집합에 있는 모든 명령어들에 대한 클래스가 되는데, 예를 들어 add, sub, mul 명령어등이 이에 해당된다고 볼 수 있다.

만약 명령어 집합을 처음부터 디자인해서 아키텍처를 새로 만든다고 가정을 하여 보자. 이러한 상황은 특수 목적에 맞는 프로세서인 ASIP (Application Specific Instruction Set Processor)를 개발할 때는 필수 불가결하게 발생하는 것이다.

이 때, 성능 향상을 위해서 현재 명령어 집합에 *mac (multiply-and-accumulate)*이라고 하는 명령어를 추가한다면, 원래의 구조를 그대로 둔 채 mac이라고 하는 클래스를 Instruction 클래스로부터 상속시켜서 만든 후, action()이라고 하는 method에 mac 연산을 기술하여 주면 된다. 즉, 기존의 모든 코드는 전혀 수정하지 않은 채 재사용성과 확장성이 확보 되는 것이다. 물론, 이러한 예는 객체 지향의 개념만 정확히 이해하면 디자인 패턴을 몰라도 되는 것이다. 이와 같이 디자인 패턴은 사람들이 흔히 아는 것부터 시작해서 거의 생각하기 힘든 것까지 여러 종류가 다양하게 있을 수 있다. 따라서, 패턴을 적절히 이용하기 위해서는 잘 정리된 데이터베이스나 카탈로그 혹은 패턴을 이용하기 쉽게 짜여진 툴이 있어야겠다.

4.2 패턴의 복합 적용 : Template Method & Strategy Pattern

이번에는 RISS의 시뮬레이션 core의 개발을 위하여 template method와 strategy 패턴이 복합되어 적용된 예를 보이겠다.

우선 template method 패턴에 대해서 간단히 기술하면 다음과 같다. Template method 패턴은 어떤 클래스 A의 operation이 변화 가능할 때, 그 operation의 구체적인 구현을 명시하지 않는 대신, 그것을 이루는 primitive operation들의 순서를 나열한다. 그리고, 이러한 primitive operation들은 A 클

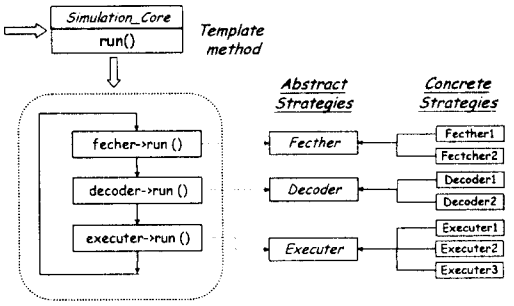

그림 3 디자인 패턴의 복합 적용 예

래스의 하위 클래스에서 구체적으로 명시해 준다. 이렇게 함으로써, 하위 클래스만을 바꾸어 줌으로서 클래스 A의 operation이 결국 바뀌어지게 된다. 이때 A 클래스의 객체를 사용하는 client 클래스는 전혀 코드의 수정을 할 필요가 없게 되어, 확장성 및 재 사용성이 생기는 것이다.

그림 3에서 보면 simulation의 한 step을 수행하여주는 *Simulation_Core* 클래스의 run() method가 바로 template method임을 알 수 있다. 즉, run()이라는 method를 기술함에 있어서 *fetcher1*->run(), *decoder2*->run() 과 같이 구체적으로 기술하지 않고, *fetcher*->run()과 같이 추상적으로 기술함으로써 *fetcher* 클래스의 하위 클래스중 어떤 것을 선택하더라도 그것에 상관없이, *Simulation_Core* 클래스의 run()이라는 method는 재 사용되어지는 것이다. 앞에서 설명한 것을 상기하면 fetcher, decoder 및 executer 클래스들은 각각 strategy 패턴을 이룬다는 것을 알 수 있다. 구체적인 예를 들면, fetcher의 하위 클래스로서는 한순간에 하나씩 읽어 오는 simple_fetcher 클래스와 한번에 여러개의 명령을 읽어오는 multiple_fetcher 클래스가 있을 수 있게 된다. 또한, decoder도 구현에 따라서 assembly_decoder, binary_decoder 및 BDD_decoder등 여러 가지 하위클래스를 임의로 만들어서 붙일 수 있게 된다. executer 또한 simple_executer, VLIW_executer, pipeline_executer등의 하위 클래스를 만들 수 있게 된다.

5. 결론

본 논문은 retargetable simulator의 개발에 디자인 패턴을 적용한 예를 보였다. RISS의 특성상 아키텍처의 변화를 수용하여야 하므로 확장 및 변경이 용이하고 재 사용성에 대한 요구가 크게 된다. 이것을 해결함에 있어서 디자인 패턴이 여러 군데에서 중요하게 사용되었음을 보였다. 디자인 패턴은 이외에도 어느 정도 이상의 복잡성 및 확장성을 요구하는 소프트웨어의 개발에 유용하게 쓰일 수 있다.

6. 참고문헌

[1] Madisetti V.K., Rapid Prototyping of Application-Specific Signal Processors : Current Practice, Challenges, and Roadmap, *IEEE ICPP'95 Workshop on Challenges in Parallel Processing*, August 1995.

[2] Mark R. Hartoog et al., Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. *Proceedings of DAC* 97, pp. 303-306, Anaheim California, 1997

[3] Bart Kienhuis et al., The Construction of a Retargetable Simulator for an Architecture Template, *6th International Workshop on HW/SW Codesign* (CODES/CASHE '98), pp 125-129, 1998

[4] Gamma et al, Design Patterns : *Elements of Reusable Object-Oriented Software.*, Addison-Wesley(1994)