제22회 한국정보처리학회 추계학술발표대회 논문집 제11권 제2호 (2004. 11)

C++에서 올바른 소프트웨어 콤포넌트 설계를 위한 Design by Contract 활용

곽종섭, 권기항
동아대학교 컴퓨터공학과
e-mail : janus04@donga.ac.kr, khkwon@daunet.donga.ac.kr

Applying Design by Contract for software component design in C++

Jongseob Kwak, Keehang Kwon
Department of Computer Engineering, Dong-A University

요 약

소프트웨어 신뢰성을 보장하기 위한 Eiffel의 Design by Contract[1,2] 기법은 프로그램 코드 안에 코드가 뜻하는 바를 함께 기술하는 것으로 소프트웨어가 명세의 주어진 조건에 따라 정확히 동작하도록 하고 있다. 그것은 재사용성이 높은 콤포넌트를 기반으로 하는 콤포넌트 기반 소프트웨어 개발방법에 있어서 중요한 특징이다. 본 논문에서는 C++언어의 타입 상속에서 올바른 의미적 타입 계층구조를 유지할 수 있도록 Design by Contract 기법을 적용하고, 기능을 활용할 수 있는 방안을 제시하였다. 또한, 객체지향 프로그래밍에 있어서 올바른 타입 구조를 형성할 수 있도록 하여, 견고한 소프트웨어 콤포넌트를 제작할 수 있도록 유도하였다.

1. 서론

콤포넌트 기반 소프트웨어 개발방법은 재사용성이 뛰어난 콤포넌트의 조합으로 새로운 소프트웨어를 제자가는 생산성이 매우 뛰어난 개발 방법이다[3]. 재사용성이 뛰어난 콤포넌트의 품질은 전체 소프트웨어에 중요한 영향을 주게 되므로 그러한 콤포넌트의 신뢰성은 무엇보다도 중요하다고 할 수 있다. 신뢰성을 보장하기 위해서는 구체적인 기능 명세가 필수적이며, 작성된 기능 명세에 따라 정확하게 동작하는 콤포넌트를 작성하는 것이 관건이 된다.

객체지향 프로그래밍 언어에서 추상 데이터 타입(Abstract Data Type)을 구현하는 수단인 클래스의 올바른 작성을 위해 대부분의 객체지향 언어들은 정적 형검사와 같은 기능을 제공하여 어느 정도 구현상의 문법적 실수를 막을 수 있도록 하고 있지만, 실행시 기능 명세에 기록되어 있지 않은 논리 오류 처리의 문제에 대한 해결책은 비교적 소홀한듯 하다. 하지만, Eiffel은 Design by Contract 기법을 언어가 직접 지원하여 기능 명세를 코드와 함께 작성하는 것으로 어느 정도 실행시 발생하는 문제점들을 사전에 검사해 볼 수 있도록 하고 있다.

Design by Contract는 구성자 또는 메써드의 실행 전과 실행 후의 조건을 제시하여 이행 유무에 따라 기능의 논리적 오류를 해결할 수 있도록 하였다.

본 논문에서는 Eiffel 언어가 지원하는 Design by Contract 기법을 C++언어에서 활용할 수 있도록 하기 위한 기존연구[4]에서 부족했던 클래스의 타입 상속기능에 대한 문제점을 보충하여 Eiffel 언어와 비슷한 수준의 소프트웨어 신뢰성을 보장 하고자 한다. 또한, 그러한 기능이 소프트웨어 설계에 대한 제약이 되어 C++언어를 이용한 개발자의 신뢰성 높은 소프트웨어 설계 및 제작을 유도하고자 한다.

2. Design by Contract의 소개

Design by Contract는 각 객체지향 언어의 클래스를 의뢰인과 공급자로 가정하여, 그것들 간의 계약에 따라 소프트웨어가 동작하는 것으로 묘사한다. 의뢰인과 공급자간에 서로가 지켜야 하고 이행해야 하는 정형화된 계약을 사전에 구체적으로 정의해야 하며, 계약은 클래스 메써드의 요구조건(precondition)과 만족조건

(postcondition) 그리고 클래스에 대한 불변조건(class invariant)을 명시함으로써 계약 위반을 쉽게 알아낼 수 있도록 하고 있다. 기능 명세인 계약을 코드와 함께 기술함으로써 실행시 논리적 오류를 쉽게 검출해 낼 수 있다.

Eiffel 에서는 아래와 같은 문법으로 요구조건과 만족조건, 클래스 불변조건을 표현한다.

```
class T
...
feature
    function is
        require
            -- preconditions
        ...
        ensure
            -- postcondition
    end
invariant
    -- class invariant
end
```

Eiffel 의 클래스 메써드의 구성에서 require 와 ensure, invariant 절은 요구조건과 만족조건, 클래스 불변조건을 나타내는 부분이다. 이것을 일반화된 표현식으로 나타내면 [그림 1]과 같다.

R1. 구성자 mkT의 적절한 인자값 x_{mkT} 에 대해서

$\{Default_T \wedge require_{mkT}(x_{mkT})\}\ Body_{mkT}\ \{ensure_{mkT}(x_{mkT}) \wedge INV_T\}$

R2. 메써드 m 와 적절한 인자값 x_m 에 대해서

$\{require_m(x_m) \wedge INV_T\}\ Body_m\ \{ensure_m(x_m) \wedge INV_T\}$

- $Default_T$: 클래스 T의 속성에 대한 기본값
- $require_m(x_m)$, $ensure_m(x_m)$: 클래스의 구성자 또는 메써드 m 에 대한 요구조건과 만족조건
- $Body_m$: 클래스의 구성자 또는 메써드 m 의 실행부
- INV_T : 클래스 T 의 불변 조건
- $\wedge$: 논리 and 연산자

[그림 1] 단일 클래스의 정확성에 대한 정의

Eiffel 언어로 작성된 BoundedStack 클래스를 통해 구체적인 사용예를 살펴보자.

```
class BounedStack[G] feature
    count : INTEGER
    ...
    feature
        push(x:G) is
            require
                not full
            -- do something (body of the push method) ···
            ensure
                not empty
                item = x
                count = old count + 1
        end
        ··· empty, full, top, pop ···
    feature {NONE}
        representation : ARRAY[G]
    invariant
        count_non_negative: 0 <= count
        count_bounded : count <= capacity
        consistent_with_array_size capacity = representation.capacity
        empty_if_no_elements : empty = (count = 0)
        item_at_top : (count > 0) implies (representation.top(count) = top)
end
```

- push 메써드의 의미

$\{require_{push}(not\ full) \wedge INV_{BoundedStack}\}\ Body_{push}\ \{ensure_{push}(not\ empty \wedge item = x \wedge count = old\ count + 1) \wedge INV_{BoundedStack}\}$

- invariant의 의미

$INV_{BoundedStack} = (0<=count \wedge count<=capacity \wedge capacity = representation.capacity \wedge empty=(count=0) \wedge (count>0)\ implies\ (representation.top(count) = top))$

3. 기존의 연구에서 제안된 방법

C++언어에서도 2 절에서 보여준 Eiffel 의 예와 같은 기능을 수행할 수 있도록 제공되는 표준기능인 assert 매크로를 이용하여 단정문을 표현할 수가 있다. 하지만, 예외상황시 고급예외처리 기능을 사용할 수가 없고, 요구조건과 만족조건을 구별할 수 있는 체계를 갖추고 있지 않아 Design by Contract 기법을 수용하기에 충분하지 않다. 이러한 문제점을 해결하기 위해 [4]에서는 다음과 같은 Assertor<T>클래스를 제안 하였다.

```
template<class G>
class BoundedStack{
    int count;
    ...
    void put(G x){
        Assertor<BoundedStack> pas(" put()" );
        pas.require(!isFull());
        // do something ···
        pas.ensure(!isEmpty());
        pas.ensure(top() == x,);
        pas.ensure(count == old().count+1);
    }
    ...
    bool invariant(){
        return 0 <= count && count <= capacity && ···;
    }
}
```

하지만, [4]에서 제안된 방법은 단일 클래스에 대한 적용은 적당하지만, 상속에 의한 계층 구조관계에 대해서는 Eiffel 언어가 지원하는 기능을 기대할 수 없다.

4. Subcontract 의 적용을 위한 Assertion 클래스

객체지향 프로그래밍언어에서의 하위타입은 상위타입의 값(value) 속성과 메써드(method)의 동작 속성을 만족해야만 한다. 그것은 상위클래스 메써드의 계약을 하위클래스 메써드가 지켜야 함을 의미한다(subcontract). 다음은 Stack[G]클래스를 상속받은 BoundedStack[G]클래스에서 subcontract 를 보여주는 Eiffel 의 구체적인 예이다.

```
class Stack[G] feature
   count : INTEGER
   ...
feature
   push(x:G) is
      -- do something (the body of push) ···
      ensure
         not empty
         item = x
         count = old count + 1
   end
   ··· empty, full, top, pop ···
feature {NONE}
   representation : ARRAY[G]
invariant
   count_non_negative: 0 <= count
   empty_if_no_elements : empty = (count = 0)
   item_at_top : (count > 0) implies (representation.top(count) =
top)
end

class BounedStack[G] feature
...
inherit
   Stack[G]
feature
   count : INTEGER
   ...
feature
   push(x:G) is
      require
         not full
      -- do something (the body of push) ···
      ensure
         not empty
         item = x
         count = old count + 1
   end
   -- empty, full, top, pop ···
feature {NONE}
   representation : ARRAY[G]
invariant
   count_bounded : count <= capacity
   consistent_with_array_size : capacity = representation.capacity
end
```

[그림 2]는 subcontract 에 대한 일반화된 표현식이다.

R1. 구성자 mkT 의 적절한 인자값 x_{mkT} 에 대해서
$\{Default_T \wedge pre_{mkT}(x_{mkT})\}Body_{mkT}\{post_{mkT}(x_{mkT}) \wedge c\_INV\}$

R2. 메써드 m 과 적절한 인자값 x_m 에 대해서
$\{[p\_pre_m(x_m) \vee c\_pre_m(x_m)] \wedge [p\_INV \wedge c\_INV]\}$
$Body_m$
$\{[p\_post_m(x_m) \wedge c\_post_m(x_m)] \wedge [p\_INV \wedge_d c\_INV]\}$

- $Default_T$: 클래스 T 의 속성에 대한 기본값
- $p\_pre_m(x_m)$, $p\_post_m(x_m)$: 부모 클래스의 메써드 m 에 대한 요구조건과 만족조건
- $c\_pre_m(x_m)$, $c\_post_m(x_m)$: 자식 클래스의 메써드 m 에 대한 요구 조건과 만족조건
- p_INV : 부모 클래스의 클래스 불변조건
- c_INV : 자식 클래스의 클래스 불변조건
- $\wedge, \vee$: 논리 and, 논리 or 연산자
- [ ] : 우선순위 연산자

[그림 2] 상속관계에서 메써드에 대한 subcontract

본 논문에서 제안하는 Assertion<T>클래스는 상속을 통해 require, ensure, invariant 멤버함수를 제공하여 Eiffel 언어가 지원하는 Design by Contract 기법과 유사한 기능을 하도록 하였다.

```
template<class T>
class Assertions : public Clonable{
   public:
      class Exception : public exception{
         public:
            Exception(const std::string& label) :
               exception((" Assertion violation: " + label
+ " ." ).c_srt()){}
      };
   private:
      static bool PreconditionEnabled;
      static bool PostconditionEnabled;
    std::map<const std::string, const T*> oldObjects;

   public:
      virtual void remember(const std::struing& tag){
         delete oldObjects[tag];
         oldObjects[tag] = dynamic_cast<const T*>(Clone());
      }
      virtual const T& old(const std::string& tag){
         return dynamic_cast<const T&>(*oldObjects[tag]);
      }
      virtual bool invariant() const{
         return true;
      }
      virtual void require(bool b, const  std::string& label) const{
         if(PreconditionEnabled && !b)
            Throw Assert::Exception(" require " + label);
      }
```

```
    virtual void ensure(bool b, const std::string& label) const{
        if(PostconditionEnabled && !invariant())
            throw Assert::Exception(" invariant " + label);
        if(PostconditionEnabled && !b)
            throw Assert::Exception(" ensure " + label);
    }
};
```

Assertion<T>클래스는 C++의 assert 매크로의 기능을 강화한 것이다. assert 매크로가 가지고 있지 않던 고급 예외처리 기능과의 연계, 요구조건과 만족조건을 구별할 수 있도록 체계화, 그리고 타입상속관계에서 올바른 타입 계층구조를 형성할 수 있도록 설계 되었다.

다음절은 Assertion<T>클래스를 활용한 실제 설계상의 이점을 보여주는 예시이다.

5. 올바른 클래스 계층구조 설계를 위한 제안

Stack<G>클래스의 하위 클래스인 BoundedStack<G> 클래스는 상위 클래스의 메써드 push 를 재정의하여 하위클래스의 상태를 보장해 줄 수 있는 단정문이 추가된 것을 볼 수 있다. 그리고 이것은 [그림 2]의 R2 규칙을 지킴으로써 클래스간의 올바른 의미적 계층구조를 형성할 수 있도록 유도한다.

```
template<class G>
class Stack : public Assertion<Stack>{
    int count;
    ...
public:
    void push(G x){
        // do something
        ensure(!isEmpty() && top() == x && count == old().count+1,
" push()" );
    }
    ...
    bool invariant(){
        return count >= 0;
    }
};


template<class G>
class BoundedStacl : public Stack{
    int count;
    int capacity;
    ...
public:
    void push(G x){
        Stack::push(x);
        require( !isFull() );
      // do something ···
        ensure(!isEmpty() && top() ==x && count == old().count+1,
" push()" );
    }
    ...
    bool invariant(){
        return (0 <= count) && (count <= capacity) && ···;
    }
};
```

앞의 예제는 Design by Contract 기법이 객체지향 프로그래밍의 서브타이핑에 의한 올바른 타입 계층 구조를 유지할 수 있도록 의미관계를 밝혀 주고 있다.

6. 결론

소프트웨어가 명세에 다라 정확히 동작한다는 것을 증명하는 대부분의 기술은 정형화된 명세언어[7]를 이용한 명세서 작성과 그에 따른 구현을 비교 검증함으로써 이루어 지고 있다. 하지만, 이러한 방법은 명세와 구현을 따로 작성해야 하며, 서로의 표현력 차이로 인해 정확한 검증의 어려움이 있다. Eiffel 의 Design by Contract 기법은 명세와 구현을 표현력이 같은 구현언어를 이용함으로써 문제를 해결하였다. 뿐만 아니라, 실행시 발생하는 논리적 오류에 빠르게 대처할 수 있기도 하다. 그러므로, C++언어와 같은 산업에서 널리 사용되는 언어에 이러한 기능을 제안하는 것은 많은 의미를 가질 수 있을 것이다.

본 논문에서는 기존 연구 [4]에서 모자랐던 타입 상속기능에 대해서도 Design by Contract 기법을 활용할 수 있도록 함으로써, C++언어에서 코드와 명세를 함께 작성하여 신뢰성이 높은 소프트웨어를 제작할 수 있는 기능을 제안하였다. 또한, 타입 계층구조의 올바른 구성을 강요하였다. 이것은 C++언어가 가지고 있는 정적 형검사와 더불어 실행시 발생할 수 있는 논리적 오류를 쉽게 찾아 낼 수 있도록 하여, 견고한 소프트웨어를 제작할 수 있도록 유도할 것이다.

이러한 기능을 사용하기 위해서는 반드시 추상 데이터 형과 같은 명세를 구체적으로 작성하는 과정이 필요하다.

참고문헌

[1] Bertrand Meyer, "Object-Oriented Software Construction", 2nd Bk&CD edition, Prentice Hall, 2000.

[2] Bertrand Meyer, "Applying Design by Contract", In Computer(IEEE), vol. 25, no. 10, pages 40-51 (1992).

[3] Alan W. Brown, "Component-Based Software Engineering: Selected Papers from the Software Engineering Institute", Wiely-IEEE Computer Society Pr, 1996.

[4] 김재우 외 4 명, "C++ 언어를 위한 체계적인 단정문 기능", 한국정보과학회, (1998).

[5] Bjarne Stroustrup, "The C++ Programming Language", 3rd edition, Addison Wesley, 1997

[6] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", Communications of the ACM, vol. 12, no. 10 (1969).

[7] Graeme Smith, "The Object-Z Specification Language", Kluwer Academic Publishers, 1999.