面向对象框架扩展约束的一种规约*

胡文蕙+, 贾 涛, 闫 哲

(北京大学 信息科学技术学院 软件研究所,北京 100871)

A Specification for Extension Constraint of Object-Oriented Framework

HU Wen-Hui+, JIA Tao, YAN Zhe

(Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

+ Corresponding author: Phn: +86-10-62761900, E-mail: huwh@cs.pku.edu.cn, http://www.cs.pku.edu.cn

Received 2004-01-06; Accepted 2004-03-29

Hu WH, Jia T, Yan Z. A specification for extension constraint of object-oriented framework. *Journal of Software*, 2004,15(Suppl.):138~142.

Abstract: This paper analyzes expandable and extended elements of Object-Oriented Framework and the relationship between them. For such elements, this paper specially discusses framework extension constraint that framework users should follow, and gives specifications of extension constraints using OCL(object constraint language).

Key words: OO framework; hot spot; extension constraint

摘 要: 分析了面向对象框架的可扩展元素和被扩展元素及其关系,针对这些元素讨论了框架使用者应该遵守的框架扩展约束,并利用 OCL(object constraint language,对象约束语言)给出了扩展约束的规约.

关键词: 面向对象框架;扩展点;扩展约束

应用框架是应用系统的整体或部分可复用的设计和实现,在一定意义上可以被看作是整个系统或子系统的骨架[1].应用开发者可以通过对框架的扩展,实现一个完整的应用.本文提到的"扩展"是基于框架实现特定应用的一种机制.

面向对象框架作为应用框架的一种实现技术,可以通过一组相互协作的类/对象来表达[1].面向对象框架的扩展点可以是设计模式(design pattern)[2]、控制流(control flow)、抽象类(abstract class)或以上元素的组合.其中设计模式表达了一类相似问题的解决方案,被实现为一组相互协作的类,其中包括具体类和抽象类.控制流是以特定顺序执行的一组方法,其目标是完成一个业务流程.抽象类至少包括一个可以被扩展的抽象方法(abstract method),在框架的扩展过程中必须实现之.在框架的扩展过程中产生的扩展实体包括扩展子类(subclass)、扩展

* Supported by the National High-Tech Research and Development Plan of China under Grant No.2001AA113171 (国家高科技发展计划(863)); the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312006 (国家重点基础研究发展规划(973)); the National Research Foundation for the Doctoral Program of Higher Education of Chinaunder Grant No.2002001016 (国家教育部博士点基金)

作者简介: 胡文蕙(1977－),女,山西省五寨人,博士生,主要研究领域为软件工程,应用框架,构件技术;贾涛(1980－),男,博士生,主要研究领域为软件工程及相关技术;闫哲(1980－),男,博士生,主要研究领域为软件工程及相关技术.

属性(extended attribute)、扩展方法(extended method)以及对抽象方法进行实例化得到的具体方法(concrete method).扩展子类是通过创建抽象类的继承子类得到的,扩展子类中包含框架使用过程中新增加的扩展属性和扩展方法以及通过实例化父类抽象方法得到的具体方法.

每一个框架都有其设计约定,当基于框架进行应用开发时,都应该遵守其中的扩展约束或限制.本文采用OCL(object constraint Language)[3]——一种面向对象模型的约束规约语言,对面向对象框架中的可扩展实体以及被扩展实体进行规约,给出它们之间的良构规则,以指导应用开发者在框架实例化的过程中正确和有效地使用框架以及提高基于框架应用的计算能力.

1 面向对象框架扩展约束规约

由以上分析可见,影响框架扩展的实体有以下 4 种:扩展点(hot-spot)、抽象类(abstract class)、抽象方法(abstract method)以及控制流(control flow).本文将每一个可扩展的实体看作是 OCL 中的一个语境(context),同时在特定的语境中讨论其扩展约束.

1.1 扩展点(Hot Spot)

语义:

扩展点是一个包(package),封装了设计模式、控制流、抽象类以及它们的组合,形成了一个应用的扩展机制.通过对其中控制流和抽象类的特化,实现一个扩展点的实例化[4].

约束:

每一个扩展点可能包括若干设计模式或者控制流,而且包括至少一个抽象类.同时每一个设计模式至少包括一个抽象类,而且必须为每一个抽象类定义并实现其扩展子类.

```
context hot-spot<hot-spot name>
    self. patterns-> ( size >= 0 and forAll (pattern. abstractClasses -> size> 0))
    self. controlFlow -> size >= 0
    self. abstractClasses -> ( size > 0 and forAll (abstractClass. subclasses-> size > 0))
    self. patterns = {< pattern name1>, <pattern name2>, ...}
    self. abstractClass = {<abstractClass name1>, <abstractClass name2>,...}
end hot-spot<hot-spot name>
```

1.2 抽象类(Abstract Class)

语义:

抽象类是扩展点的核心组成部分,对抽象类进行扩展主要是为其创建扩展子类,并根据应用的特定需求实现之.具体实现扩展子类的过程中可能存在以下两类扩展行为:(1)实现抽象类定义的抽象方法;(2)增加新的属性和方法.

约束:

(1) 抽象类的子类必须具体实现父类定义的所有抽象方法

```
context abstractClass <abstractClass name>
    self. abstractMethod = {<method signature1>, <method signature2>,...}
    self. subclasses -> forAll ( abstractMethods -> forAll (abstractMethod.
        instantiation = true))
end abstractClass <abstractClass name >
method signature := <method name> '(' <parameter list> ')'
parameter list := [<empty> | <parameter> | <parameter> ',' <parameter list>]
parameter := <parameter name> ':' <data type>
```

(2) 抽象类的每一个子类必须增加的扩展属性以及其可见性约束

```
context abstractClass <abstractClass name>
    self. subclasses-> forAll (extendedAttributions-> exists (type = <data type>
        and visibility = <TypeOfVisibility>))
```

以上 OCL 语句表示,必须为类<abstractClass name>的所有扩展子类增加一个数据类型为<data type>的属

性,而且必须定义其可见性为<TypeOfVisibility>.其中 data type 的取值可以是简单数据类型,也可以是框架中定义的类或结构等复杂数据类型;TypeOfVisibility 的取值可以是 private、protected 或者 public.

有些时候为了保证框架扩展的正确性,要求将扩展子类中增添的新方法和新属性定义为私有的,这样只有该扩展子类自身的方法能够访问扩展方法和属性,而其它类则无权访问.这样的约束虽然可能减弱扩展的灵活性,但是可以有效减少由于无法控制扩展类和框架类之间不可预见的交互而导致的错误.其规约如下:

context abstractClass <abstractClass name >
  self. subclasses -> **forAll** (extendedAttributions->**forAll** (visibility = private))

(1) 抽象类的每一个子类必须增加的扩展方法及其可见性约束

context abstractClass <abstractClass name>
  self. subclasses -> **forAll** (extendedMethods -> **exists** (<method signature> **and**
    visibility = <TypeOfVisibility>))

(2) 与扩展属性的可见性约束类似,可以分别规约每一个新增加的扩展方法的可见性,也可以将所有扩展方法组成的集合作为一个整体来规约其可见性,如下:

context abstractClass <abstractClass name>
  self. subclasses -> **forAll** (extendedAttributions->**forAll** (visibility = private))

1.3 抽象方法(Abstrct Method)

语义:

抽象方法是指抽象类中已经定义的但是还没有实现的方法,在实现抽象类的继承子类(subclass)的过程中必须实现之.抽象方法的实现就是应用系统的特殊功能需求的实现.

约束:

在扩展子类中实现父类定义的抽象方法时,不能破坏父类中的假设前提,因此,需要规约抽象方法的前置和后置条件.例如,父类 classA 定义的抽象方法 int methodA ()的返回值必须大于 300,那么该抽象类的继承子类在实现抽象方法 methodA 的过程中也要遵循以上后置条件,规约如下:

context abstractMethod <subclass(classA):: method(): int>
  pre: true
  post: result>300

下面给出规约抽象方法前置条件和后置条件的一般语法:

context abstractMethod (<class name> / subclass(<class name>)::<method signature>))
  pre: <expression1>
  post: <expression2>

其中 expression 的语法如下:

expression := [**true** | **false** | <operation_expression> op1 <operation_expression>]
operation_expression := [<identifier > | <number> | <identifier> op2 <(expression)> |
    <number> op2<expression> | <expression> op2 <expression>]
number := [实数| 字符串]
identifier := [result | 方法所在类中的属性标识符 | 方法参数列表中的参数标识符]
op1 := ['>' | '<' | '=' | '>=' | '<=']
op2 := ['+' | '-' | '*' | '/']

在抽象方法的实现过程中,并不允许随意调用其它类提供的方法,例如不同方法对某一状态的改变结果可能产生冲突,因此需要给出对不同方法的调用应该满足的约束.以图 1 中类 Observer 的抽象方法 void update(Subject & s)为例,描述 Observer 的子类在实现 Observer::update(Subject & s)时应该满足的方法调用约束.因为方法 Subject::notify()的实现需要调用方法 Observer::update(Subject & s),因此 Observer::update (Subject & s)的实现不能调用对象 s 中的 notify()方法,避免引发死循环.规约如下:

context abstractMethod (Observer::update(Subject & s))
  self. callMethods -> forAll (!s->notify() = **true**)
end abstractMethod (Observer::update(Subject & s))

下面给出规约在抽象方法的实现中对于其它方法的调用需要满足的约束语法:

context abstractMethod (<class name> / subclass(<class name>) :: <method signature>)

self. callMethods -> **forAll** (<boolean_expression>) = **true**)

其中 boolean_expression 是逻辑表达式,其运算子为方法原型,其语法如下:

boolean-expression := [<method signature> | <method signature > '||' <boolean-expression>|
<method signature> '&' <boolean-expression> | '!'< boolean-expression>]
method signature := [<class name> '::' <method name> '('<parameter list>')' ':' <data type>|
<object name> '.' <method name> '('<parameter list>')' ':' <data type>

parameter list 的定义同 3.2 节.

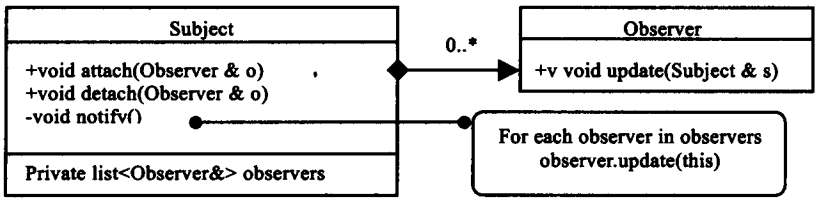

图 1 Observer 设计模式的类图[2]

1.4 控制流(Control Flow)

语义:

控制流是以特定顺序执行的一组方法,用以完成一个业务流程[4].控制流的扩展就是利用已实现的具体方法(包括框架固定点提供的具体方法、扩展点已定义的由应用开发者实现的抽象方法以及应用开发者新增加的扩展方法),遵循一定的约束条件,创建满足应用特殊需求的业务流程.

约束:

在控制流的扩展约束中主要规约方法调用的顺序、方法调用必须遵守的约定或者满足的条件,例如强调必须在某一点前或后调用某一方法.利用以下三种控制流类型来构造控制流约束:; (顺序)、if(condition) {}(条件选择)以及 loop {}(循环).图 2 给出了 3 种控制流构造元素的语义,其中 a 和 b 表示对特定方法的调用和执行.a;b 意味着 a 与 b 是顺序执行的,a 无条件到达 b; if(condition) {a}意味着 a 必须由 if 条件语句封装;loop {a}表示循环调用方法 a.

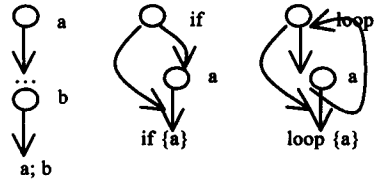

图 2 控制流类型

(1) a; b

context <context name>
self. flowType = **#sequence** (<method signatureA>; <method signatureB>)

(2) if (condition){a}

context <context name>
self. flowType = **#condition**(**if** <condition> **then** <method signatureA>)
condition := [<constraint statement> | <boolean-expression> |<expression> |
<condition> '&&' <condition> | <condition> '||' <condition> | '!'<condition>]

其中 method signature 的定义同 3.3 节.constraint statement 可以是前文给出的任意扩展约束语句,boolean-expression 的定义同 3.2 节,expression 的定义同 3.3 节.

(3) loop {a}

context <context name>
self. flowType = **#loop** (<method signatureA>)

值得注意的是,有些控制流约束并不是针对某一个特定的方法或者类,它的作用域是整个框架,而且在框架的整个实例化过程中都是有效的,因此控制流约束的语境可以是特定的方法、类或框架.

(4) 组合控制流约束

可以对以上类型的控制流进行组合和递归形成更复杂的控制流约束.例如,当图 1 中对象 Subject 调用了方法 void attach(Observer &)或者 void detach(Observer &),即改变了对象 Subject 的状态,此时要通知所有的 Observer 对象,该操作由 Subject::notify()来完成,Subject:: notify()会依次调用每一个 Observer 对象的 update()方法.以上约束的规约就要用到**#condition** 和**#loop** 的组合:

```
context abstractMethod (Subject::notify():void)
    self. controlFlow->#condition (if Subject. callMethod ->forall
        (attach (Observer &):void || detach (Observer &): void ) = true) then
        loop (Observer:: update(observer &))
end abstractMethod (Subject::notify():void)
```

2 结束语

一些研究工作给出了不同的面向对象框架描述方法,例如"菜谱"方法[5]、模式方法[6]以及基于 UML[7]扩展描述方法等,这些描述方法主要强调框架的设计以及如何使用框架,而对框架的扩展约束考虑较少.本文通过对面向对象框架中可扩展元素的分析,表达了其扩展点的构成本质;针对框架的可扩展构造元素和扩展元素,在已有描述方法的基础上利用 OCL 给出了框架扩展约束规约,并在电信综合营业系统框架的开发和使用中得到应用.以上工作可以指导面向对象框架的构造,并且有助于应用开发者更加有效地使用框架,同时可以作为对基于框架开发的应用系统进行一致性和正确性验证的基础.其中有关对应用系统进行验证的技术和方法还有待于进一步研究.

References:

[1] Fayad M, Schmidt DC. Object-Oriented application frameworks. Communications of the ACM, 1997,40(10):43~54.

[2] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. New York; Addison-Wesley, 1995. 175~184, 315~324.

[3] Jos Warmer, Anneke Kleppe, The Object Constraint Language: Precise Modeling with UML. New York, Addison-Wesley, 1998

[4] Hu WH, Zhao W, Zhang SK, Wang LF. Study of application framework meta-model based on component technology. Journal of Software, 2004,15(1):1~8 (in Chinese with English abstract). http://www.jos.org.cn/ 1000-9825/15/1.htm

[5] Krasner GE, Pope ST. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object Oriented Programming, 1988.

[6] Johnson RE. Documenting frameworks with patterns. In: Proc. of the OOPSLA'92. Vancouver, Canada, 1992.

[7] Fontoura M, Pree W, Rumpe B. UML-F: A modeling language for object-oriented frameworks. In: Proc. of the 14th European Conf. on Object Oriented Programming (ECOOP 2000). Lecture Notes in Computer Science 1850, Springer, 2000. 63~82.

附中文参考文献:

[4] 胡文蕙,赵文,张世琨,王立福.基于构件技术的应用框架元模型的研究.软件学报,2004,15(1):1~8. http://www.jos.org.cn/1000-9825/15/1.htm