Когай Анна Ростиславовна

УДК 004.432.4

ПРИМЕНЕНИЕ АСПЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ ДЛЯ ПОДДЕРЖКИ ТЕХНОЛОГИИ DESIGN-BY-CONTRACT

Аннотация

В работе рассматривается технология контрактного проектирования (Design-by-Contract), предлагающая систематический подход к спецификации и реализации классов и их взаимосвязей в программной системе. Исследуются преимущества использования технологии с точки зрения повышения надежности программного обеспечения. Для поддержки технологии предлагается использовать методологию аспектно-ориентированного программирования. Приведены основные принципы спецификации контрактов и примеры их реализации с помощью аспектов в системе Aspect.NET.

Ключевые слова: design-by-contract, DBC, aspect-oriented programming, AOP, Aspect.NET.

ВВЕДЕНИЕ

Разработка программ с использованием объектно-ориентированного подхода [1] является в настоящее время широко распространённым и, вероятно, основным видом деятельности программиста. Возможность применения объектного подхода доказана для задач самого разного характера. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что уменьшает степень риска разработки сложных систем и повышает уверенность в правильности принимаемых решений. В настоящее время объектно-ориентированное проектирование – единственная методология, позволяющая справиться со сложностью, присущей очень большим системам.

Необходимо заметить, что объектно-ориентированный подход предполагает повышенные требования к качеству – прежде всего, для обеспечения возможности повторного использования программных компонентов, в корректности которых не должно быть никаких сомнений.

Для обеспечения уверенности в надлежащей работе объектно-ориентированного программного обеспечения необходим *систематический подход* к спецификации и реализации классов и их взаимосвязей в программной системе [2]. Такой подход существует и называется «Контрактное проектирование» («Design by contract») [3]. В его рамках программная система рассматривается в виде множества взаимодействующих компонентов, чьи отношения строятся на основе точно определённой спецификации взаимных обязательств – контрактов. Контрактные спецификации и процесс проектирования на их основе были введены Бертраном Мейером (Bertrand Meyer) в 1986 году в контексте разработки языка программирования Eiffel [4]. Ключевые идеи контрактного подхода стали составной частью языка Eiffel, который следует рассматривать не только как конкретный язык программирования, но и как метод разработки программного обеспечения.

Помимо Eiffel встроенная поддержка технологии design-by-contract имеется в языках программирования Chrome [5], D [6], Lisaac [7], Nemerle [8] и др. Что касается языков без прямой поддержки, при

попытке применять принципы контрактного проектирования обычными средствами возникают определённые трудности. В данной работе рассмотрена концепция *аспектно-ориентированного* программирования [9] как одно из возможных решений данной проблемы.

ОБЗОР ТЕХНОЛОГИИ КОНТРАКТНОГО ПРОЕКТИРОВАНИЯ

Зададимся вопросом: что означает утверждение «программный элемент корректен»? По Мейеру [10] программная система или её элемент сами по себе не могут быть ни корректны, ни некорректны. Корректность подразумевается лишь по отношению к некоторой спецификации. То есть термин «корректность» не применим к программному элементу, он имеет смысл лишь для пары – «программный элемент и его спецификация». Значит, первым и важнейшим шагом на пути к корректности является разработка спецификаций программных элементов – точных и чётких определений того, что, собственно, этот программный элемент должен уметь делать. Спецификации должны разрабатываться одновременно с написанием программы, а лучше – до её написания. Конечно, наличие спецификации вовсе не гарантирует, что программа будет работать в полном соответствии с ней. Однако без спецификации, определяющей, что должен делать модуль, вероятность того, что он всё-таки именно это и будет делать, очень мала. Среди следствий такого подхода можно отметить следующие:

- Программная система с самого начала проектируется так, чтобы быть корректной. В момент написания программа снабжается аргументами, характеризующими её корректность.
- Значительно более полное и ясное представление проблемы, возможных путей её развития.
- Упрощение задачи создания программной документации.
- Обеспечение основ для систематического тестирования и отладки.

Теория контрактного проектирования предполагает приложение спецификации к каждому программному элементу. Эти спецификации (или «контракты») управляют взаимодействием элемента с окружающим его миром.

Что же представляет собой контрактная спецификация? Центральная метафора подхода заимствована из бизнеса – компоненты программной системы взаимодействуют друг с другом на основе взаимных *обязательств (obligations)* и *выгод (benefits)*. Контракт – это набор *утверждений (assertions)*, которые чётко описывают, что должен и не должен делать каждый конкретный метод. Утверждения могут быть трёх видов: предусловия, постусловия и инварианты.

Если компонент-поставщик предоставляет окружению (клиентам) некоторую функциональность, он может наложить *предусловие (precondition)* на её использование. Предусловие определяет выгоду для поставщика и обязательство для клиентов. Компонент-поставщик, в свою очередь, может гарантировать выполнение некоторого действия с помощью *постусловия (postcondition)*, которое определяет обязательство для него и выгоду для клиентских компонентов. *Инвариант (invariant)* класса – это утверждение, выражающее общие согласованные ограничения, применимые к каждому экземпляру класса. Инвариант применяется к классу как целому, и этим отличается от предусловий и постусловий, характеризующих отдельные программы (методы).

Предусловия выражают ограничения, выполнение которых необходимо для корректной работы метода. Предусловия применяются ко всем вызовам программы как внутри класса, так и у клиента. Корректная система никогда не вызовет программу в состоянии, в котором не выполняется её предусловие.

Постусловие определяет состояние, завершающее выполнение программы. Постусловие гарантирует, что выполнение программы приводит к состоянию с заданными свойствами в предположении, что

программа была запущена в состоянии, удовлетворяющем предусловию.

Инвариант – это множество утверждений, которым удовлетворяет каждый экземпляр класса во все «стабильные» времена:

– на момент создания экземпляра, то есть сразу после выполнения конструктора;

– перед вызовом и после вызова любого метода данного класса. Вполне возможно, что некоторый метод в процессе своей работы разрушает инвариант, но, завершая работу, он обязан его восстановить.

Неявно инвариант можно рассматривать как добавления к предусловиям и постусловиям каждой программы класса. Поэтому может показаться, что понятие инварианта класса избыточно – это часть пред- и постусловий методов. Однако здесь важен именно глубокий смысл инварианта, выходящий за пределы отдельных программ. Ведь инвариант применим не только к уже написанным программам класса, но и к тем, которые ещё будут написаны. Добавление, удаление и изменение функциональности – явление частое и нормальное. В этом изменчивом процессе инварианты отражают фундаментальные соотношения, характерные для класса.

Таким образом, контракт является спецификацией класса, которая точно описывает, какие услуги предоставляет класс.

ПРИМЕНЕНИЕ КОНТРАКТНОГО ПРОЕКТИРОВАНИЯ В ЯЗЫКАХ БЕЗ ПРЯМОЙ ПОДДЕРЖКИ С ПОМОЩЬЮ МЕТОДОЛОГИИ АСПЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ (АОП)

Встроенная поддержка технологии контрактного проектирования присутствует далеко не во всех языках программирования. В то же время при попытке соблюдать контракты обычными средствами в языках без прямой поддержки возникают следующие проблемы [11]:

- Неспособность программных компонентов к повторному использованию. Контрактные условия могут быть верны для компонентов только в контексте конкретной системы. В другой системе вполне могут быть другие соглашения, что ведёт к необходимости модификации кода.
- Код проверки утверждений контракта перемешивается с основным кодом компонентов. Это снижает «читабельность» кода и, как следствие, повышает вероятность появления ошибок.
- Код проверки утверждений контрактов рассредоточивается по всей системе. Если возникнет потребность изменить какое-либо из условий, то придётся произвести изменения во всех модулях, на которые оно распространяется.
- Громоздкая реализация проверки утверждений. Изъятие, вставка и модификация проверок – нормальное явление, но при этом достаточно тяжело поддерживать систему в согласованном состоянии.

Проверка утверждений контракта – типичный пример сквозной функциональности, реализация которой присутствует во многих программных модулях. Классический объектно-ориентированный подход предоставляет удобные средства для выделения логики программы в отдельные компоненты. Но в рамках данного подхода не существует возможности локализовать в отдельные модули функциональность, которая пронизывает всю систему.

Из-за того, что реализация сквозной функциональности не может быть обособлена средствами языка программирования в отдельном программном модуле, элементы этой реализации присутствуют в том или ином виде в большинстве компонентов, образующих программную систему. На рис. 1*а* схематично изображено распределение функциональности проверки контрактов по компонентам некоторой программной системы. В каждой компоненте отмечен код, реализующий проверку некоторых утверждений. Распределение той же функциональности в некотором идеальном варианте показано на рис. 1*б*.

Одним из существующих решений данной проблемы является аспектно-ориентированное программирование (АОП). Аспектно-ориентированный подход предлагает средства инкапсуляции сквозной функциональности в отдельных программных модулях – аспектах. Аспектные модули могут затрагивать многие компоненты и используют так называемые точки вставки (*join points*) для реализации регулярных действий, которые в обычном случае рассредоточены по всему тексту программы. Механизм описания логики сквозной функциональности и механизм описания точек программы, в которых данная логика будет применяться, зависит от конкретной реализации АОП.

Ключевой составляющей любого инструмента аспектно-ориентированного программирования является так называемый аспектный интегратор, или компоновщик (*weaver*). Он реализует автоматическую компоновку аспектных модулей и традиционных компонентов программы. Автоматизированная компоновка аспектов и компонентов является мощным средством генерации кода и гарантирует, что аспект будет применен ко всем компонентам, которые он затрагивает, чего сложно добиться, если вносить сквозную функциональность в компоненты вручную.

Аспектно-ориентированный подход, очевидно, решает проблемы, связанные с запутанным и рассредоточенным кодом, и преимущества от его использования очевидны [12]. Данный подход предоставляет простое и мощное решение для реализации проверки проектных контрактов.

В настоящее время наиболее известным и широко используемым инструментом аспектно-ориентированного программирования является AspectJ [13]. Он представляет собой расширение языка Java и нацелен на работу исключительно с Java-платформой. Однако реализация концепций АОП только на уровне языковых расширений представляется неполной и неэффективной. Более перспективным стоит считать проникновение самих базовых идей АОП и их использование на уровне платформ (.NET, Java) и технологий (COM, CORBA) по аналогии с уже традиционным использованием ООП.

С этой точки зрения особого внимания заслуживает система Aspect.NET [14], разработанная в Санкт-Петербургском государственном университете при поддержке Microsoft Research. Aspect.NET принципиально отличается от AspectJ, прежде всего, самим подходом к реализации концепций АОП. Aspect.NET не является расширением какого-то конкретного языка программирования, не вводит в язык новых конструкций, как это сделано в AspectJ. Аспекты описываются с помощью метаязыка (*meta-language, ML*), после чего конвертор (*Aspect.NET.ML converter*) преобразует эти инструкции в специальные АОП атрибуты, которыми помечаются классы и методы, являющиеся частью аспектной логики. Таким образом, аспект в системе Aspect.NET – это обычный класс, снабжённый некоторой описательной информацией (атрибутами), которая позволяет пользователям и системе определить, что данный класс следует рассматривать как аспект. В процессе компиляции АОП-атрибуты преобразуются в метаданные аспектной сборки и хранятся вместе с кодом на промежуточном языке платформы .NET (*MSIL, Microsoft Intermediate Language*). Такой подход делает систему

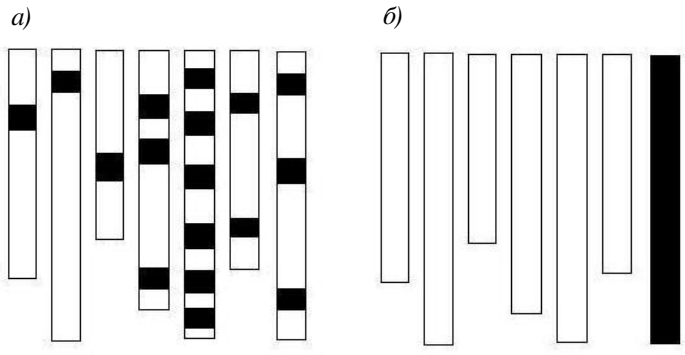

Рис. 1. Распределение сквозной функциональности:
а) по компонентам некоторой программной системы;
б) в идеальном варианте

Aspect.NET «языково-независимой», и, хотя на данный момент конвертор поддерживает только преобразование из метаязыка в язык C#, в будущих релизах планируется добавить поддержку других языков платформы .NET.

Кроме того, в отличие от многих других инструментов аспектно-ориентированного программирования, Aspect.NET предоставляет возможность при необходимости вручную откорректировать список возможных точек вставки (*join points*), используя графический пользовательский интерфейс (*GUI, Graphic User Interface*). Это свойство оказывается весьма удобным при проверке проектных контрактов. Разработчик имеет возможность комбинировать по своему усмотрению те проверки, которые он хотел бы осуществить. Например, проверять только предусловия, обрабатывать предусловия и постусловия или же отключить обработку утверждений. Такого рода мониторинг утверждений позволяет легко проверить, действительно ли программа делает то, что от неё ожидает разработчик. Это является простым, но мощным средством обнаружения ошибок.

ПРИНЦИПЫ СПЕЦИФИКАЦИИ КОНТРАКТОВ И ИХ РЕАЛИЗАЦИЯ С ПОМОЩЬЮ АСПЕКТОВ В СИСТЕМЕ ASPECT.NET

В качестве примера внедрения контрактных спецификаций в программы, написанные на языке C#, рассмотрим класс *Stack*, моделирующий работу со стеком – структурой с политикой доступа Last – In, First – Out. Приведённые наблюдения и рассуждения несложны, но, как отмечает сам автор теории контрактного проектирования Б. Мейер [10], таковы все научные результаты – они начинаются с обычных наблюдений и продолжаются путем простых рассуждений, но всё это нужно делать упорно и настойчиво (листинг 1).

Прежде всего, следует выделить семантические свойства класса, не зависящие от специфики реализации, и на основе этого разработать спецификацию. Для класса Stack такими свойствами являются:

1. Метод *Push* не может быть вызван, если стек заполнен.

2. Метод *Pop* не может быть применён к пустому стеку.

3. После завершения работы метода *Push* стек не может быть пуст, на его вер-

Листинг 1

```
public partial class Stack
    {
         private int capacity; // max number of stack elements
         private int count;    // current number of stack elements
         private int[] representationArray; // stack representation

          public bool IsEmpty()
            {
                return (count == 0);
            }
            public bool IsFull()
            {
                return (count == capacity);
            }

            public void Push(int element)      // Add element on top
            { ... }
            public int Pop()      // Get top element
            { ... }
        }
```

шине находится только что добавленный элемент, число элементов стека увеличилось на 1.

4. После завершения работы метода *Pop* стек не может быть полон, метод возвращает элемент с вершины стека, число элементов стека уменьшилось на 1.

Стоит обратить особое внимание на методы *IsEmpty* и *IsFull*. В терминах класса *Stack* постусловие для этих методов должно проверять, является ли возвращаемое значение результатом сравнения count == 0 (или count == capacity). Но ведь внутри метода именно это значение и было передано оператору return. Закономерен вопрос: в чем же смысл написания постусловия? Не является ли оно избыточным?

Мейер [10] дает однозначный ответ на этот вопрос: между двумя указанными конструкциями большая разница. Присваивание – это команда (инструкция), в результате выполнения которой изменяется состояние. Утверждение же ничего не делает, оно лишь специфицирует *свойство* ожидаемого заключительного состояния.

То, что две нотации – присваивание и эквивалентность – оказались так близки, не должно затемнять их фундаментальное различие. Инструкция является частью реализации, утверждение – элементом спецификации. Клиенты модуля обычно интересуются утверждениями, а не реализациями. При переходе к более сложным примерам различие между спецификацией и реализацией будет возрастать.

Таким образом, нами определён контракт для класса *Stack*. Значит, разработчик класса при реализации его методов смело может предполагать, что все ограничения, заданные предусловием, выполняются; ему нет нужды проверять их в теле метода. Так, например, для метода *Push* можно смело предполагать, что стек не полон, поскольку это предусмотрено предусловием. В дополнительной проверке не только нет необходимости, но это и недопустимо с точки зрения методологии контрактного проектирования. Это правило можно сформулировать следующим образом: *ни при каких обстоятельствах в теле программы не должно проверяться её предусловие (non-redundancy principle)* [10].

С этой точки зрения теория контрактного проектирования противоречит принципам «защитного программирования» (*defensive programming*) [15]. Его идея заключается в том, что для обеспечения надёжности каждая программа должна защищать себя настолько, насколько это возможно – лучше больше проверок, чем недостаточно, дополнительная проверка в худшем случае будет бесполезной. Проектирование по контракту утверждает обратное – избыточные проверки могут нанести вред.

Конечно, если ограничить видение проблемы узким миром единственной программы, кажется, что включение дополнительной проверки делает программу более устойчивой. Но мир системы не ограничивается одной программой – он содержит множество программ во множестве классов. Для получения надёжной системы необходимо перейти к макроскопическому видению проблемы, обобщающему всю архитектуру.

С этой глобальной точки зрения *простота* становится критическим фактором. В своей монографии [10] Мейер, исследуя базисные механизмы надёжности, делает вывод: «самый большой враг надёжности (и качества программного обеспечения в целом) – это сложность. Создавая наши структуры настолько простыми, насколько это возможно, мы достигаем необходимого (но не достаточного) условия, гарантирующего надёжность. Также необходим (но не достаточен) постоянный акцент на создание элегантного и читабельного программного обеспечения. Программные тексты не только пишутся, они ещё читаются и переписываются по много раз. Ясность и простота нотации языковых конструкций – основа любого изощрённого подхода к надёжности».

Итак, контракт для класса *Stack* определён, и можно переходить к его реализации в виде аспектного модуля. Определение точек вставки для проверки предус-

Листинг 2

```
%before %call *Stack.Push(int) && %args(..)
     %action
          public static void CheckPushPrecondition(int element)
          { ... }

     %before %call *Stack.Pop()
     %action
          public static void CheckPopPrecondition()
          { ... }
```

ловий методов *Push* и *Pop* может выглядеть следующим образом (см. листинг 2).

Немного сложнее обстоит дело с постусловиями и вот почему. В языках со встроенной поддержкой контрактного проектирования в постусловиях доступна специальная конструкция *old <выражение>*. Она позволяет получить доступ к значению, которое данное выражение имело на входе программы. Соответственно, если выражению не предшествует old, имеется в виду значение на выходе программы.

Поскольку в нашем арсенале нет подобной функциональности, оказывается недостаточным просто написать листинг 3 по той простой причине, что внутри метода нет никакой возможности получить оба состояния вызывающего объекта – *на входе* и *на выходе* программы.

Однако это алгоритмическое затруднение может быть разрешено следующим образом (см. листинг 4).

Аналогично – для метода *Pop()* (см. листинг 5).

В случае, когда результатом выполнения метода является некоторое возвращаемое значение, например, как в методах *IsFull* и *IsEmpty*, можно воспользоваться специальной конструкцией *RetValue* (доступна в версии Aspect.NET 2.1) (листинг 6).

ЗАКЛЮЧЕНИЕ

Соблюдение несложных принципов контрактного проектирования является основой для решения многих проблем, критических для объектно-ориентированного подхода: на какие понятия опираться на стадии анализа, как специфицировать

Листинг 3

```
     %after %call *Stack.Push(int) && %args(arg[0])
     %action
          public static void CheckPushPostcondition(int element)
          { ... }
```

Листинг 4

```
%instead %call *Stack.Push(int) && %args(arg[0])
     %action
          public static void CheckPushPostcondition(int element)
          {
               Stack originalStack = (Stack)TargetObject;
               originalStack.Push(element);
               Stack resultStack = originalStack;
               // сравнение originalStack и resultStack
          }
```

Листинг 5

```
%instead %call *Stack.Pop()
     %action
          public static void CheckPopPostcondition()
          {
               Stack originalStack = (Stack)TargetObject;
               originalStack.Pop();
               Stack resultStack = originalStack;
               // сравнение originalStack и resultStack
          }
```

компоненты, чем руководствоваться при выполнении тестирования. Перечисленные преимущества приобретаются за счёт явного и чёткого разделения обязанностей между классами и их пользователями.

Это, в первую очередь, приводит к систематизации и упрощению проектирования. Процесс становится для участников более прозрачным и понятным.

Значительно улучшается ясность программных текстов, что обуславливает лучшее понимание кода как самими разработчиками, так и другими участниками технологического процесса: аналитиками, создателями тестов по методу белого ящика. Результат – более полные и эффективные процедуры тестирования и отладки.

Контракты существенно упрощают процесс создания программной документации, так как спецификации класса, описывающие свойства его атрибутов и методов, сами по себе являются хорошей документацией.

Становится более приемлемым повторное использование класса – как за счёт улучшенной документации, обеспечивающей корректное использование класса, так и за счёт возможности проверки корректности применения класса в новых условиях.

Таким образом, принципы контрактного проектирования в совокупности с удобным инструментом аспектно-ориентированного программирования предоставляют весьма продуктивный подход к тестированию, отладке и, следовательно, повышению качества программного обеспечения, где поиск ошибок ведётся не вслепую, а на базе условий, сформулированных самими разработчиками. И надёжная система получается как итог надлежащим образом встроенных в процесс разработки действий.

Листинг 6

```
%after %call *Stack.IsEmpty()
     %action
     public static void CheckIsEmptyPostcondition()
     {
          Stack originalStack = (Stack)TargetObject;
          if ((bool)RetValue != (originalStack.count == 0))
          { … }
     }
```

Литература

1. *Г. Буч.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, 1997.

2. *B. Meyer.* Systematic Concurrent Object-Oriented Programming. Communicatons of the ACM, Vol. 36, №. 9, September 1993. P. 56–80.

3. *Б. Мейер.* Построение надёжного объектно-ориентированного программного обеспечения: введение в контрактное проектирование. Открытые системы (http://www.osp.ru), № 6, 1998.
4. http://www.eiffel.com – Eiffel Software web site.
5. http://www.remobjects.com/product/page.asp?id={E10F7F5C-AE94-4833-9E4B-2EDD5ED69768} – Chrome Programming Language web site.
6. http://www.digitalmars.com/d – D Programming language web site.
7. http://isaacproject.u-strasbg.fr/li.html – Lisaac Programming Language web site.
8. http://nemerle.org/ – Nemerle Programming Language web site.
9. http://www.aosd.net – Aspect-Oriented Software Development web site.
10. *B. Meyer.* Object-Oriented Software Construction. Prentice Hall, 1997, p. 1296.
11. *В. Павлов.* Анализ вариантов применения аспектно-ориентированного подхода при разработке программных систем. СПб, изд-во СПбГЭТУ (ЛЭТИ), 2004.
12. *Г. Полличе.* Аспектно-ориентированное программирование (АОП): для чего его лучше использовать? http://www.ibm.com/developerworks/ru/
13. http://www.aspectj.org – The AspectJ web site.
14. *V. Safonov.* Aspect.NET 2.1 user guide. SPbSU, 2007.
15. *С. Макконел.* Совершенный код. СПб.: Питер, 2005, 896 с.

Abstract

The article deals with Design-by-Contract technology which provides a systematic approach to specifying and implementing object-oriented software elements. Advantages of the technology with a view to software reliability are discussed. Aspect-Oriented Programming methodology is reviewed and proposed as a method for Design-by-Contract support. Basic concepts of specifying contracts are stated and examples are provided on how to implement contracts by the use of aspects in Aspect.NET system.

Keywords: design-by-contract, DBC, aspect-oriented programming, AOP, Aspect.NET.

Когай Анна Ростиславовна, аспирантка кафедры информатики математико-механического факультета СПбГУ, инженер-программист T-Systems CIS, anna.kogay@gmail.com.