УДК 004.43

ИССЛЕДОВАНИЕ ВОЗМОЖНОСТИ ПРИМЕНЕНИЯ НЕСКОЛЬКИХ ПАРАДИГМ ПРОГРАММИРОВАНИЯ В НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

© 2016 Л.В. Яблокова

Самарский национальный исследовательский университет имени академика С.П. Королёва

Статья поступила в редакцию 11.11.2016

В статье предложены и исследованы способы конструирования прикладного программного обеспечения для научно-исследовательской деятельности, основанные на применении нескольких парадигм программирования и соответствующих им паттернов проектирования. Постановку задачи для исследования можно сформулировать следующим образом: необходимо проанализировать основные парадигмы программирования, сформировать представление о способах и мотивации к их применению. Для каждой парадигмы нужно с использованием одного из поддерживающих ее языков представить сценарий иллюстрирующий ситуацию, при которой возможно применение, как самой парадигмы, так и соответствующих ее уровню паттернов. Как результат исследования необходимо определить и применить такую стратегию разработки программ для физико-математических расчетов, которая была бы способна удовлетворить высоким требованиям, предъявляемым к качеству программного обеспечения, создаваемого в рамках научно-исследовательской работы. Исследование проводилось с целью формализации представлений о средствах решения проблем связанных со сложностью, являющейся следствием необходимого уровня гибкости программ, который нужно поддерживать для проведения математических расчетов в области теоретической физики.

Ключевые слова: парадигмы программирования, императивное программирование, процедурное программирование, объектно-ориентированное программирование, обобщённое программирование.

ВВЕДЕНИЕ

Информатика, как технологическая наука, должна по определению включать прагматику, а это, в свою очередь, должно предполагать постоянное сравнение ее составляющих, в соответствии с их правильностью, полезностью и актуальностью. Как известно, количественной мерой любого результата является оценка, основанная на соответствующих критериях и нормах, которые действуют в рамках размерности, определяемой природой их ценности. А прагматическая природа информатики предполагает, что остается и используется только самое эффективное. Все остальное либо выбраковывается, либо переосмысливается и, в конечном счете, обновляется. Чего же мы ждем от наших программ, предназначенных для научных исследований, чего хотим от конкретной реализации в частности? Мы хотим, чтобы программа работала правильно и надежно, но кроме этого, нам нужно, чтобы она была легко адаптируема к новым условиям, которые часто меняются в ходе решения той или иной задачи. Одним из критериев качества программы является то, насколько внимательно в процессе проектирования авторы отнеслись к процессу формализации типичных взаимодействий между ее компонентами. Если на этапе проектирования таким механизмам уделить достаточное внимание, то архитектура программы получится более компактной, простой и понятной, а код более структурированным и адаптируемым к вновь специфицируемым требованиям.

Яблокова Людмила Вениаминовна, старший преподаватель кафедры прикладной математики.
E-mail: lyablokova@gmail.com

Представленные в статье результаты исследования с помощью небольших примеров кода на мультипарадигменном языке MATLAB показывают, как используя несколько парадигм программирования и соответствующих им паттернов проектирования, можно создавать прикладные программы для физико-математических расчетов.

МЕТОДИКА ИССЛЕДОВАНИЯ

Современные языки среднего и высокого уровня, в большинстве своем, поддерживают парадигму процедурного программирования. Акцент в ней делается на средстве обработки информации – алгоритме, необходимом для выполнения требуемых вычислений. Правило для процедурного стиля программирования можно сформулировать так: «Decide which procedures you want; use the best algorithms you can find». Парадигма объектно-ориентированного программирования, являясь развитием процедурного стиля, уже использует иерархии полиморфных типов данных, связанных отношением наследования. Правило для такого стиля программирования можно определить, так: «Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance». Обобщённое программирование, как развитие

процедурного и объектно-ориентированного направлений, также определяет свой уровень абстракции. Алгоритм, написанный в обобщённом стиле, может применяться для любых типов, удовлетворяющих синтаксическим и семантическим требованиям, которые он предъявляет к своим аргументам. Правило для обобщённого стиля программирования звучит так: «Decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures».

Далее приводится краткое описание основных парадигм программирования и упрощенные фрагменты кода на нескольких языках, в той или иной степени используемых в научно-исследовательской работе и демонстрирующих применение этих парадигм и соответствующих им паттернов проектирования.

Императивное программирование (ИП):

1. Процесс работы или план вычислений описывается в виде последовательности инструкций, изменяющих состояние программы.

2. Основное понятие – оператор.

3. Основной принцип – принцип последовательного изменения состояния программы пошаговым образом.

Файл main.f90 (FORTRAN):

```
program main
    implicit none
    print *, 'Hello from FORTRAN!';
end program main
```

Процедурное программирование (ПП):

1. Частный случай императивной парадигмы.

2. Задается последовательность процедур, каждая из которых есть последовательность элементарных действий или вызовов таких же процедур.

3. В качестве базовых элементов используются данные и процедуры.

4. Данные не зависят от процедур.

Файл main.c (C):

```
#include <stdio.h>

int main()
{
        printf("Hello! This is %s.", "procedural
programming");
        return 0;
```

Объектно-ориентированное программирование (ООП):

1. Основные понятия – класс объектов, структура данных и тип значений.

2. Рассматривает процесс обработки информации как обработку объектов.

3. Позволяет выполнять модификации программы посредством объявления новых подклассов.

4. Полиморфное поведение упрощает логику управления и на уровне кода можно не заботиться о распознавании принадлежности определенного метода конкретному классу, находящемуся на соответствующем уровне иерархии.

Пример использования паттерна Dependency Injection (Java).

1. Файл IPrintHelperImpl.java:

```
public interface IPrintHelperImpl
{
        void printValue(Object arg);
};
```

2. Файл PrintHelper.java:

```
public class PrintHelper
{
        private IPrintHelperImpl impl_ = null;

        public void printValue(Object arg)
        {
                this.impl_.printValue(arg);
        }
        public PrintHelper(IPrintHelperImpl
impl)
        {
                this.impl_ = impl;
        }
```

3. Файл PrintFormattedImpl.java:

```
public class PrintFormattedImpl implements IPrintHelperImpl
{
        private String format_ = null;

        public void printValue(Object arg)
        {
                System.out.printf(this.format_, arg);
        }
        public PrintFormattedImpl(String format)
        {
                this.format_ = format;
        }
};
```

4. Файл EntryPoint.java:

```
public class EntryPoint
{
        public static void main(String[] args)
        {
                String format = "%s This is Java!";
                IPrintHelperImpl impl = new
PrintFormattedImpl(format);

                PrintHelper printer = new PrintHelper(impl);
                printer.printValue(«Hello!»);
        }
```

Пример использования концепции мультиметодов (C#).

1. Файл IPrinter.cs:

```
public interface IPrinter
{
    void Print(Formatter formatter);
};
```

2. Файл Printer.cs:

```
public abstract class Printer : IPrinter
{
    protected object value_ = null;
    public abstract void Print(Formatter formatter);
    public virtual object Value
    {
        get
        {
            return this.value_;
        }
    }
};
```

3. Файл BooleanPrinter.cs:

```
using System;

public class BooleanPrinter : Printer
{
    public override void Print(Formatter formatter)
    {
        Console.WriteLine(formatter.Format(this));
    }
    public BooleanPrinter(bool value)
    {
        this.value_ = value;
    }
};
```

4. Файл DoublePrinter.cs:

```
using System;

public class DoublePrinter : Printer
{
    public override void Print(Formatter formatter)
    {
        Console.WriteLine(formatter.Format(this));
    }
    public DoublePrinter(double value)
    {
        this.value_ = value;
    }
};
```

5. Файл StringPrinter.cs:

```
using System;

public class StringPrinter : Printer
{
    public override void Print(Formatter formatter)
    {
        Console.WriteLine(formatter.Format(this));
    }
    public StringPrinter(string value)
    {
        this.value_ = value;
    }
};
```

6. Файл Formatter.cs:

```
using System;

public abstract class Formatter
{
    public virtual string Format(BooleanPrinter printer)
    {
        throw new InvalidOperationException();
    }
    public virtual string Format(DoublePrinter printer)
    {
        throw new InvalidOperationException();
    }
    public virtual string Format(StringPrinter printer)
    {
        throw new InvalidOperationException();
    }
};
```

7. Файл BooleanFormatter.cs:

```
using System;

public class BooleanFormatter : Formatter
{
    public override string Format(BooleanPrinter printer)
    {
        bool value = Convert.ToBoolean(printer.Value);
        return string.Format("This is C#; boolean value = {0}.", value);
    }
};
```

8. Файл DoubleFormatter.cs:

```
using System;

public class DoubleFormatter : Formatter
{
    public override string Format(DoublePrinter printer)
    {
        double value = Convert.ToDouble(printer.Value);
        return string.Format("This is OOP; double value = {0}.", value);
    }
};
```

9. Файл StringFormatter.cs:

```
using System;

public class StringFormatter : Formatter
{
    public override string Format(StringPrinter printer)
    {
        string value = Convert.ToString(printer.Value);
        string format = "This is multimethods; string value = '{0}'.";
        return string.Format(format, value);
    }
};
```

10. Файл Program.cs:

```
class Program
{
    static void Main(string[] args)
    {
        IPrinter printer1 = new BooleanPrinter(true);
        printer1.Print(new BooleanFormatter());

        IPrinter printer2 = new DoublePrinter(19.92);
        printer2.Print(new DoubleFormatter());

        IPrinter printer3 = new StringPrinter("SSAU");
        printer3.Print(new StringFormatter());

        IPrinter printer4 = new DoublePrinter(20.15);
        printer4.Print(new BooleanFormatter());
        //Error:  InvalidOperationException
    }
};
```

Обобщённое программирование (ОП):

1. Суть в таком описании данных и алгоритмов, которое можно применять к различным типам данных и сигнатурам алгоритмов не меняя, при этом, само описание.

2. Поддержку обобщённого программирования обеспечивают шаблоны.

3. Генерирование всех необходимых экземпляров шаблонных элементов программы возлагается на компилятор.

4. Повторное использование кода для библиотек и алгоритмов.

Пример с использованием частичной специализации (C++).

1. Файл main.cpp:

```
#include “print_helper.hpp”

int main(int argc, char *argv[])
{
        print_helper<IOSTREAM_TAG, int> printer1;
        printer1.print_value(21);

        print_helper<STDIO_TAG, double> printer2(“Real number:
%.3f”);
        printer2.print_value(21.235123);

        return 0;
```

2. Файл print_helper.hpp:

```
#ifndef PRINT_HELPER_HPP_
#define PRINT_HELPER_HPP_

#include<stdio.h>
#include<iostream>

struct STDIO_TAG;
struct IOSTREAM_TAG;

template<typename PRINT_TAG, typename ARG>
class print_helper;

template<typename ARG>
class print_helper<STDIO_TAG, ARG>
{
private:
        const char *format_;
public:
        void print_value(ARG const &arg)
        {
                printf_s(format_, arg);
        }
        print_helper(char const *format) : format_
(format){}
};

template<typename ARG>
class print_helper<IOSTREAM_TAG, ARG>
{
public:
        void print_value(ARG const &arg)
        {
                std::cout << arg << std::endl;
        }
};
```

Для каждой парадигмы характерны ассоциированные с ней приоритеты для оценки качества программирования, отличия в инструментах и методах работы и соответственно – стиле мышления, и используемых стереотипах. Но ключевым моментом при любом стиле программирования был и остается уровень абстракции, который позволяет рассматривать абстрактные сущности как способ описания контекста предметной области.

РЕЗУЛЬТАТЫ ИССЛЕДОВАНИЙ

Механизм внедрения зависимостей (Dependency Injection) – это набор принципов и паттернов проектирования программного обеспечения, которые дают возможность разрабатывать слабосвязанный код.

1. Файл Solution.m:

```
classdef Solution < handle
    properties(Access = private)
        impl_;
    end
    methods
        function this = Solution(impl)
            this.impl_ = impl;
        end
        function solve(this, solver)
            this.impl_.solve(solver);
```

2. Файл SolutionImpl.m:

```
classdef(Abstract = true) SolutionImpl < handle
    methods(Abstract = true)
        solve(this, solver);
    end
```

3. Файл SolutionParams.m:

```
classdef(Abstract = true) SolutionParams < handle
    properties(Constant = true)
        Lv = 2.99792458e14;
    end
    properties(SetAccess = protected)
        Q;
        Qt;
        QT;
        Lambda;
```

4. Файл MaxwellSolution.m:

```
classdef MaxwellSolution < SolutionImpl & SolutionParams
    properties(Constant = true)
        Mu_0 = 1.2566370614e-12;
        Epsilon_0 = 8.8541878174e-18;
    end
    properties(Dependent = true, SetAccess = private)
        Lz;
        Nt;
        Nz;
        Ht;
        Hz;
        C1;
        C4;
        C5;
    end
    methods
        function this = MaxwellSolution(q, qt, qT, lambda)
            %Initialization of protected class members
        end
        function solve(this, solver)
            solver.solveMaxwell(this);
        end
```

5. Файл D_AlembertSolution.m:

```
classdef D_AlembertSolution < SolutionImpl & SolutionParams

    properties(Dependent = true, SetAccess = private)

        Lz;

        Nt;

        Nz;

        Ht;

        Hz;

        C1;

        C5;

    end

    methods

        function this = D_AlembertSolution(q, qt, qT, lambda)

            %Initialization of protected class members

        end

        function solve(this, solver)

            solver.solveD_Alembert(this);

        end

        %Property accessor methods
```

Двойная диспетчеризация (Double Dispatch) – это способ организации кода на базе паттерна Посетитель (Visitor), позволяющий расширить интерфейс базового класса путем создания отдельной иерархии для виртуализации операций. В обобщенном виде подобную реализацию можно представить в виде матрицы, строки которой соответствуют всевозможным типам возможных решений, а столбцы – типам решателя. В каждой ячейке матрицы находится конкретный алгоритм для обработки сочетания типов. Отсутствие типизации в MATLAB делает невозможным перегрузку операции решения с указанием типов конкретных решателей, поэтому изменение сигнатуры приходится производить за счет смены имен операций.

6. Файл Solver.m:

```
classdef(Abstract = true) Solver < handle

    methods(Abstract = true)

        solveMaxwell(this, solution);

        solveD_Alembert(this, solution);

    end
```

7. Файл MaxwellSolver.m:

```
classdef MaxwellSolver < Solver
    methods
        function solveMaxwell(~, solution)
            amp = exp(0.0 * 1i);
            src = fix(solution.Nz / 2);
            ex = zeros(1, solution.Nz);
            hy = zeros(1, solution.Nz - 1);
            %Solving Maxwell equations
            %        .
            %        .
            %        .
        end
        function solveD_Alembert(~, solution)
```

```
classdef MaxwellSolver < Solver
    methods
        function solveMaxwell(~, solution)
            amp = exp(0.0 * 1i);
            src = fix(solution.Nz / 2);
```

8. Файл D_AlembertSolver.m:

```
classdef D_AlembertSolver < Solver

    methods

        function solveMaxwell(~, solution)

            error('Invlid operation for solution %s with solver %s',...

            class(solution), 'D_AlambertSolver');

        end

        function solveD_Alembert(~, solution)

            a = exp(0.0 * 1i);

            ex = zeros(1, solution.Nz);

            extt = zeros(1, solution.Nz);

            exttt = zeros(1, solution.Nz);

            rad_locat = solution.Nz - solution.Q;

            %Solving D_Alembert equation

            %        .

            %        .
```

9. Файл Solve.m:

```
clear solve;


solution1 = Solution(MaxwellSolution(1, 2, 3, 4));

solution1.solve(MaxwellSolver());


solution2 = Solution(D_AlembertSolution(1, 2, 3, 4));

solution2.solve(D_AlembertSolver());


solution3 = Solution(MaxwellSolution(1, 2, 3, 4));
```

ЗАКЛЮЧЕНИЕ

Программирование – это искусство выражать решения задач так, чтобы компьютер мог их осуществить. С другой стороны программы нужно создавать для других людей, стараясь обеспечить их высокое качество. Программирование – это нечто большее, нежели простое следование некоторым правилам, предписаниям и чтению справочника в поисках подходящей языковой инструкции. Чтобы программный код получился хорошим исследователю-программисту необходимо понимать основные идеи, принципы и методы, определяющие общие концепции и конструкции языков программирования. Как и математика, программирование представляет собой полезное интеллектуальное упражнение, оттачивающее мыслительные способности. Как и любая научно-исследовательская деятельность программирование – это постоянное экспериментирование с данными, со способом их обработки, представления и интерпретации, проверка предположений и гипотез о возможности применения того или иного подхода в рамках решаемой задачи и используемого языка программирования. Разные парадигмы подразумевают собственные принципы абстракции, общие для множества

методик разработки. Каждая методика – это свой способ группировки абстрактных представлений в соответствии с их общими свойствами, включая и закономерности в отличиях отдельных объектов. Результат определения общности должен указывать на повторяющиеся внешние свойства системы, характерные для выбранной предметной области. Но, с другой стороны, выявленная общность помогает упорядочить неявную структуру предметной области, обнаруживаемую при помощи анализа в повторяющихся решениях. Использование нескольких парадигм в научно-исследовательской работе подталкивает нас обращаться к обеим точкам зрения. Анализ общности и изменчивости – это основа выбранного подхода, поскольку именно таким образом человеческий разум может создавать абстракции. А хорошая техника абстрагирования, учитывая сложность программного обеспечения и реального мира, помогает упорядочивать знания и синтезировать более качественные результаты при решении научных задач.

СПИСОК ЛИТЕРАТУРЫ

1. *Кун Т.* Структура научных революций. С вводной статьей и дополнениями 1969 г. М.: Прогресс, 1977. 300 с.
2. *Stroustrup B.* The C++ Programming Language. U.S. Addisson-Wesley, 2013. 1368 p.
3. *Chivers I.* Introduction to Programming with Fortran. Switzerland: Springer, 2012. 621 p. DOI 10.1007/978-0-85729-233-9.
4. *Фаулер М.* Рефакторинг. Улучшение существующего кода. Санкт-Петербург: Символ-Плюс, 2003. 432 с.
5. *Макконнелл С.* Совершенный код. М.: Питер, 2007. 893 с.
6. *Гамма Э.* Приемы объектно-ориентированного проектирования. М.: Питер, 2006. 368 с.
7. *Головашкин Д. Л., Яблокова Л.В.* Совместное разностное решение уравнений Даламбера и Максвелла. Одномерный случай // Компьютерная оптика. 2012. Т. 36, №4. С. 527-534.
8. *Golovashkin D.L., Vorotnikova D.G.* Long vectors algorithms for solving grid equations of explicit difference schemes // Computer Optics. 2015. Vol. 39(1). P. 87-93.
9. MATLAB. The Language of Technical Computing . URL: http://www.mathworks.com/help/matlab/index.html (дата обращения: 12.03.16).

STADY OF THE USE OF SEVERAL PARADIGMS IN SCIENTIFIC RESEARCH WORK

© 2016 L.V. Yablokova

Samara National Research University named after Academician S.P. Korolyov

The article proposed and investigated ways of designing software applications for research activities based on the use of multiple programming paradigms and their respective design patterns. Formulation of the problem for the study can be summarized as follows: you need analyze the main programming paradigms, to form an idea of the methods and motivation to their use. For each paradigm needed using one of its supporting languages to imagine a scenario illustrating the situation in which the use is possible of both the paradigm and its respective level patterns. As a result, research is needed to identify and implement such a strategy for the development of programs of physical and mathematical calculations that would be able to meet the high requirements for the quality of the software produced in the framework of research work. The study was conducted with the aim of formalizing the notions of means of addressing the problems associated with the complexity which is a consequence of the necessary level of programming flexibility, which should be supported to carry out mathematical calculations in the field of theoretical physics.

Keywords: programming paradigms, imperative programming, procedural programming, object-oriented programming, generic programming.

Lyudmila Yablokova, Senior Lecturer at the Applied Mathematics Department. E-mail: lyablokova@gmail.com