преподаются дисциплины, в рамках которых студенты изучают объектно-ориентированное программирование либо на уровне теории, либо в области разработки информационных систем, хранящих и обрабатывающих информацию в базах данных. Подобный подход формирует у студентов младших курсов представление об объектно-ориентированном программировании либо как о бесполезной для практического применения парадигме, либо как о парадигме, имеющей сугубо специфическую область использования. И хотя на старших курсах, когда студенты сталкиваются с практическими задачами в рамках курсовых и дипломных работ, это представление об объектно-ориентированной парадигме у студентов меняется, времени для формирования систематизированных знаний в данной области уже не остается.

Вместе с тем, уже с младших курсов студенты специальностей «Прикладная математика» и «Прикладная информатика» изучают различные дисциплины, в рамках которых разрабатывают приложения, решающие практические задачи, например: вычислительные методы алгебры, алгоритмы и структуры данных и т.д. При создании в рамках этих дисциплин приложений, решающих простые задачи практической направленности, как правило, студентам разрешается использовать те языки программирования и парадигмы, которые для них более удобны. В таком случае многие обучающиеся выбирают средства, более всего им понятные, и в подавляющем большинстве случаев выбор остается за процедурной парадигмой.

В последнее время на кафедре прикладной математики и механики в рамках проведения лабораторных занятий по дисциплине «Вычислительные методы алгебры» к разрабатываемым студентами приложениям предъявляется требование: использовать объектно-ориентированную парадигму. Подобный подход позволяет параллельно с получением знаний, умений и навыков по вычислительным методам алгебры закрепить знания, умения и навыки по объектно-ориентированной парадигме, полученные в курсе «Программирование».

Для студентов были разработаны методические рекомендации по применению объектно-ориентированной парадигмы, но в рамках этой дисциплины предлагаемая студентам структура классов упрощена [2–3]. Это делается для того, чтобы не концентрировать внимание студентов на программной архитектуре в ущерб самим алгоритмам вычислительной алгебры, поскольку дисциплина «Вычислительные методы алгебры» преподается на II курсе, когда студенты еще не готовы воспринимать сложные архитектурные решения, применяемые в крупных проектах, связанных с разработкой программного обеспечения. В таких проектах усложнение структуры классов и объектов обусловлено необходимостью повысить эффективность работы команды разработчиков. Обучающиеся же выполняют лабораторные работы индивидуально, поэтому и применяемые при объектно-ориентированном проектировании подходы могут быть упрощены. Однако на старших курсах можно предлагать более сложные архитектурные идеи. Для этого, а также для установления междисциплинарных связей и преемственности дисциплин студентам можно предложить продолжить использовать объектно-ориентированное программирование в других курсах. Однако в то же время необходимо переработать методические рекомендации, во-первых, адаптировав их к условиям конкретных дисциплин, во-вторых, усложнив структуру классов, максимально приблизив их к условиям крупных реальных проектов.

Целью данной работы является выработка структуры интерфейсов и классов, которая позволит ознакомить студентов с принципами объектно-ориентированного проектирования, применяемыми при разработке архитектуры сложных программных систем. Для сохранения преемственности дисциплин в нашем исследовании предлагается дальнейшее развитие объектной модели, применяемой для обработки матриц в курсе «Вычислительные методы алгебры».

Материал и методы. Основными методами исследования является объектно-ориентированный анализ и проектирование задач вычислительной алгебры, конкретнее, операции над матрицами.

В курсе «Вычислительные методы алгебры» студентам на лабораторных работах предлагается реализовать различные методы решения систем линейных алгебраических уравнений. Для хранения в памяти матриц коэффициентов уравнений применяются различные классы, учитывающие особенности некоторых методов решения систем, основанных на специальном виде матриц (верхне- и нижне-треугольные, симметричные, трехдиагональные и др.). При этом специализированные классы учитывают особенности матриц для оптимизации использования памяти. Одновременно с этим классы, выполняющие хранение матриц в памяти, организованы в иерархию, связанную отношением наследования. Однако такая иерархия лишает структуру приложения определенной гибкости.

Если рассматривать базовый класс в иерархии классов, хранящих матрицы, как поставщик полей для своих подклассов, определяющий способ хранения данных в классе, то в самом классе, как произвольной матрице, необходимо описывать поле-ссылку на произвольный двумерный массив и два целочисленных поля – размерности этого двумерного массива. Но подобный способ хранения данных в классе не отвечает по смыслу требованиям класса, хранящего, например, диагональную матрицу. Для нее имеет смысл хранить только элементы на главной диагонали в виде одномерного массива, а для единичной матрицы классу достаточно вообще только одного поля, хранящего размерность матрицы.

Таким образом, иерархию наследования классов для хранения матриц с точки зрения оптимизации способа хранения данных в полях имеет смысл «перевернуть», поставив в вершину иерархии единичную матрицу. Но в этом случае очень тяжело отслеживать логику использования данных классов. Кроме того, нарушается один из важных принципов в объектно-ориентированном проектировании – принцип подстановки Лисков [4].

В качестве решения такой задачи можно предложить строить иерархию на основе множественных отношений различных классов матриц. Этот способ построения иерархии делает логичным использование методов классов данной иерархии, но приводит либо к дублированию кода при определении и инициализации полей, либо к избыточному описанию полей, часть из которых в некоторых классах использоваться не будет.

С точки зрения принципов объектно-ориентированного проектирования [4], зависимость между классами должна строиться через интерфейсы (принцип инверсии зависимостей). То есть класс, реализующий, например, некоторый общий алгоритм решения системы линейных алгебраических уравнений, не зависящий от вида матрицы (метод Гаусса), не должен зависеть от некоторого конкретного класса, хранящего некоторую матрицу. Такой класс должен хранить (или принимать в качестве параметра метода) только ссылку на интерфейс, не владея информацией о типе конкретного объекта. Для реализации подобного подхода необходимо описать интерфейс для всех классов, хранящих информацию о матрицах. Для того чтобы не загромождать примеры, рассмотрим только квадратные матрицы (примеры предлагаются на языке программирования Java):

```
public interface Matrix {
  // получить размерность
  int size();
  // получить элемент
  / по индексам
  double get(int i, int j);
  // записать элемент
  void set(int i, int j,
           double value);
}
```

При описании классов, реализующих данный интерфейс, уже появляется определенная свобода в организации иерархии классов. Например, поскольку все классы мы будем использовать либо через интерфейс Matrix, либо без ссылки на некоторый базовый класс, можно без ущерба для соблюдения принципа подстановки Лисков использовать следующую иерархию:

```
// обычная квадратная матрица
public class SquareMatrix
          implements Matrix {
  private double a[][];
  public SquareMatrix(int
                        n) {
    // инициализация массива
    a = new double[n][n];
  }
  public int size() {
    return a.length;
  }
  public double get(int i,
                    int j) {
    return a[i][j];
  }
  public void set(int i,
                  int j,
                  double value) {
    a[i][j] = value;
  }
}

// симметричная матрица
public class SymmetricMatrix
          implements Matrix {
  protected double a[][];
  public SymmetricMatrix(int
                        n) {
    // инициализация массива
    a = new double[n][];
    for(int i = 0; i < n;
                      i++) {
      a[i] = new double[n-i];
    }
  }
  public int size() {
    return a.length;
```

```
  }
  public double get(int i,
                    int j) {
    if(i <= j) {
      return a[i][j];
    } else {
      return a[j][i];
    }
  }
  public void set(int i,
                  int j,
                  double value) {
    if(i <= j) {
      a[i][j] = value;
    } else {
      a[j][i] = value;
    }
  }
}

// верхне-треугольная матрица
public class
            TopTriangleMatrix
      extends SymmetricMatrix
      implements Matrix {
  public TopTriangleMatrix(
                     int n) {
    super(n);
  }
  public double get(int i,
                    int j) {
    if(i <= j) {
      return a[i][j];
    } else {
      return 0;
    }
  }
  public void set(int i,
                  int j,
                  double value) {
    if(i <= j) {
      a[i][j] = value;
    } else {
      if(value != 0) {
        throw new
                  Exception();
      }
    }
  }
}
```

В приведенном выше примере используются не единая иерархия, а фактически две раздельные иерархии, связанные только лишь интерфейсом. Такие независимые иерархии не увеличивают степень связанности между классами, что облегчает модификацию исходного кода при его сопровождении в дальнейшем. Также, по аналогии с этим примером, можно добавлять другие классы, не связанные с указанными, включенные в свою иерархию, но реализующие интерфейс Matrix. Это позволит таким классам «выглядеть» как матрицы, но при этом совершенно по-другому строить свое внутреннее содержимое (например, для сильно разреженных матриц).

Однако в примере выше имеются и недостатки, на которые можно не обращать особого внимания в этих классах, но которые станут существенными для других классов. Речь идет о реализации метода set() в классе TopTriangleMatrix. Поскольку класс создан для хранения верхнетреугольных матриц, то попытка записи элемента, стоящего под главной диагональю и отличного от нуля, приведет к генерированию исключительной ситуации. Конечно, конкретно в данном классе другое поведение будет неуместным. Но если рассмотреть класс, который будет хранить единичную матрицу, то реализация метода set() становится непонятной с точки зрения логики:

```
// единичная матрица
public class EMatrix
       implements Matrix {
  private int size;
  public SymmetricMatrix(
                    int n) {
    this.size = n;
  }
  public int size() {
    return size;
  }
  public double get(int i,
                    int j) {
    if(i == j) {
      return 1;
    } else {
      return 0;
    }
  }
  public void set(int i,
                  int j,
                  double value) {
    if(i == j &&
       value != 1 ||
       i != j &&
       value != 0) {
      throw new Exception();
    }
  }
```

```
}
```

Зачем давать возможность выполнения некоего действия, которое можно совершить фактически только одним способом, и при этом ожидать, что внешний вызов, инициирующий это действие, может попытаться выполнить его неверным способом? Сама по себе единичная матрица – матрица, которая не должна изменять своего значения. При попытке изменить значение некоторого элемента единичной матрицы – экземпляра класса EMatrix – необходимо заменять матрицу экземпляром уже другого класса, реализующего интерфейс Matrix. Такую задачу мы рассмотрим чуть ниже, здесь же пока можно сказать о бессмысленности метода set() для класса EMatrix, но при этом не реализовывать этот метод мы не можем, так как он описан в интерфейсе Matrix. Подобная ситуация является примером нарушения принципа разделения интерфейсов [4]. Этот принцип говорит о том, что класс не должен зависеть от тех методов интерфейса, которые он не должен реализовывать. Для приведения указанного примера в соответствие с упомянутым принципом необходимо убрать из интерфейса Matrix метод set(), перенеся его в некий интерфейс MutableMatrix, который будет расширять интерфейс Matrix. Такое разделение интерфейсов позволит классам SquareMatrix, SymmetricMatrix, TopTriangleMatrix реализовать интерфейс MutableMatrix, в то время как класс EMatrix, как и другие неизменяемые матрицы, могут реализовывать только интерфейс Matrix. При этом таким классам не будет вменяться в обязанность реализация не нужного им метода set().

Проанализируем теперь, каким образом с экземплярами описанных классов можно совершать различные действия. Для начала рассмотрим простейшие операции над матрицами, которые можно реализовать непосредственно с применением разработанных классов и интерфейсов. Возьмем операции сложения и вычитания матриц, перемножение матриц, умножение матрицы на число, обращение и транспонирование матриц. Каждую из этих операций удобно представить отдельным классом, что будет соответствовать принципу единственной ответственности [4], согласно которому для любого класса должна быть только одна причина его изменения. Возникают вопросы: каким образом реализовывать эти классы и нужно ли их связывать какими-либо отношениями? Очень часто различные операции над схожими операндами (в данном случае матрицами) реализуются с применением шаблона проектирования Command [5]. Однако применение этого шаблона сопряжено с описанием интерфейса (или абстрактного класса) для всех операций. В данном интерфейсе описывается метод, выполняющий операцию. Для матриц можно попробовать описать интерфейс:

```
public interface Operation {
  Matrix calculate(Matrix a,
                   Matrix b);
}
```

Но такой интерфейс может применяться в качестве базового только для бинарных операций, оба операнда в которых являются матрицами. Для унарных операций (обращение и транспонирование) и бинарных операций с нематричными операндами (умножение матрицы на число) этот подход использован быть не может. С другой стороны, даже если рассматривать только бинарные операции над матрицами, такое выделение операций в отдельную иерархию приведет к утрате оптимизации по хранению данных в памяти. Например, если мы вычисляем сумму двух симметричных матриц, результат также будет симметричной матрицей. Но в методе, реализующем операцию сложения, мы не владеем информацией о типе матрицы. Соответственно, для того чтобы получить результат, нам нужно в общем случае создать некоторую универсальную матрицу и записать в ее элементы вычисленный результат. Таким образом, кроме вычисления нужных элементов матрицы, в обязанности этого класса будет входить создание объекта, в котором нужно сохранять результат. Подобное поведение приводит к понижению степени зацепленности класса [4]. Однако и это еще не все. Еще одним существенным минусом предлагаемого подхода является невысокая производительность полученного решения.

Во-первых, при вычислении сложных выражений с использованием матриц большой размерности придется хранить значительный объем вспомогательных данных для промежуточных вычислений. В некоторых случаях это бывает удобно для уменьшения нагрузки на центральный процессор, так как избавляет нас от необходимости дважды вычислять одни и те же результаты. Например, рассмотрим некоторое матричное выражение (1):

$$Z = \left(A^{-1} - \lambda E\right)X + \left(A^{-1} - \lambda E\right)^{-1} Y, \quad (1)$$

здесь A, X, Y – матрицы размерности $n \times n$, E – единичная матрица размерности $n \times n$, λ – дей-

ствительное число. При вычислении этого выражения можно вычислить некоторую матрицу (2):

$$B = A^{-1} - \lambda E \,. \quad (2)$$

Затем исходное выражение принимает вид (3):

$$Z = BX + B^{-1}Y \,. \quad (3)$$

В таком случае вычисление матрицы B не будет производиться дважды, что позволит сэкономить вычислительные ресурсы.

Но на практике при решении конкретных задач хранение промежуточных результатов не всегда позволяет ускорить вычисление матричного выражения, но дополнительный объем памяти, при этом довольно значительный, используется всегда. Для решения практических задач, связанных с обработкой матриц большой размерности (в которых размерность n превышает тысячу), лучше иметь возможность сохранения промежуточных результатов, но не хранить их все.

Во-вторых, вычисление значения любой операции происходит в одном методе calculate() интерфейса Operation. Такой подход затрудняет распараллеливание вычисления операции, что не позволяет оптимизировать вычисления для многопроцессорных вычислительных систем. При этом вычисление матричных выражений для матриц большой размерности, например, на кластерных вычислительных системах, позволяет увеличить производительность. Системы с разделяемой памятью (в отличие от систем с общей памятью, таких, как многоядерные процессоры) характеризуются тем, что каждый узел вычислительной системы имеет свой центральный процессор и свою оперативную память [6]. Взаимодействие между узлами подобной системы предполагает обмен данными по специальным каналам передачи данных. Для организации эффективных вычислений на этом кластере необходимо учитывать большое количество факторов, таких, как соотношение скорости передачи данных по сетевым каналам и скорости обработки этих данных на каждом узле. В случае необходимости поддержки таких вычислений необходимо при реализации метода calculate() для каждой операции реализовывать распределенные вычисления, совмещая логику самой операции (сложение, умножение, обращение и т.д.) с управлением распределенными вычислениями. Поэтому для того чтобы изменить способ вычисления одной и той же операции, необходимо менять конкретный класс, реализующий интерфейс Operation. Подобное поведение нарушает принцип открытости–закрытости [4], согласно которому класс должен быть закрыт для изменений (то есть класс не должен изменяться, если не меняется основная логика обработки данных), но открыт для расширения (то есть класс должен позволять своим подклассам добавлять определенные свойства базовому классу через операцию наследования).

Результаты и их обсуждение. Мы рассмотрели недостатки одного из часто применяемых в объектно-ориентированном программировании способов организации однотипных действий через создание отдельного интерфейса для этих действий.

Предложим альтернативный способ организации выполнения действий над матрицами. Вернемся к интерфейсу Matrix. Данный интерфейс фактически определяет способ доступа к элементу матрицы. При этом, в зависимости от класса, реализующего этот интерфейс, элемент может извлекаться из памяти, где он хранится, либо вычисляться, например, для единичной матрицы. Так как все операции с матрицами вычисляются поэлементно, а большинство операций предполагает независимое вычисление каждого элемента результирующей матрицы, можно говорить о том, что любая операция также может реализовывать интерфейс Matrix. В этом смысле операцию можно ассоциировать с матрицей, получаемой в результате. Рассмотрим для примера операцию умножения двух матриц:

```
public class
        MatrixMultiplication
         implements Matrix {
  private Matrix a, b;
  public MatrixMultiplication
       (Matrix a, Matrix b) {
    if(a.size() ==
       b.size()) {
      this.a = a;
      this.b = b;
    } else {
      throw new Exception();
    }
  }
  public double get(int i,
                    int j) {
    double element = 0;
    for(int k = 0,
            n = a.size();
        k < n; k++) {
      element += a.get(i, k)
               * b.get(k, j);
    }
```

```
    return element;
  }
}
```

В приведенном примере мы получаем возможность вычислить произведение двух матриц, рассчитывая отдельно каждый элемент этой матрицы, что дает нам возможность использования «ленивых» (или отложенных) вычислений. Такой подход носит название шаблонов проектирования Composite (для бинарных операций, в которых две матрицы-операнда фактически рассматриваются как одна матрица-результат) или Decorator (для унарных операций, модифицирующих исходную матрицу) [5]. Но при подобной архитектуре класс, реализующий операцию, не сохраняет вычисленное значение, всего лишь возвращая его вызывающему методу. При этом класс, содержащий вызывающий метод, может самостоятельно организовать хранение полученного результата. В частности, если рассмотреть пример вычисления матричного выражения (1) в предположении, что, по аналогии с классом `MatrixMultiplication`, реализованы классы `MatixAddition`, `MatrixSubtraction`, `MatrixByNumberMultiplication`, `InverseMatrix`, можно реализовать процесс сохранения промежуточных результатов:

```
// объявление входных данных
Matrix A, X, Y;
double λ;
// ввод начальных значений в
// данном примере опустим
// создание матрицы для
// хранения промежуточного
// результата
MutableMatrix B =
  new SquareMatrix(A.size());
// создание матрицы-
// вычислителя
Matrix calculator =
       new MatrixSubtraction(
  new InverseMatrix(A),
  new
MatrixByNumberMultiplication(
    λ, new EMatrix(A.size())
  )
);
for(int i = 0, n = B.size();
    i < n; i++) {
  for(int j = 0; j < n;
                      j++) {
    B.set(i, j,
       calculator.get(i, j));
  }
}
// создание матрицы для
// хранения конечного
// результата
MutableMatrix Z =
  new SquareMatrix(A.size());
// создание матрицы для
// вычислителя
calculator =
          new MatrixAddition(
  new MatrixMultiplication(
    B, X
  ),
  new MatrixMultiplication(
    new InverseMatrix(B), Y
  )
);
for(int i = 0, n = Z.size();
                    i < n; i++) {
  for(int j = 0; j < n;
                         j++) {
    Z.set(i, j,
       calculator.get(i, j));
  }
}
```

Однако в полученном примере сохранение результата приводит в двух местах к фактическому дублированию кода, что вызывает проблему при дальнейшей поддержке программы. Идея хранения полученных промежуточных данных для их последующего повторного использования носит название механизма кэширования и может быть реализована в отдельном классе. При этом сам класс должен получать данные извне, сохранять их в свое внутреннее поле, а также предоставлять доступ к таким данным как к обычной матрице. Для этого описывается класс, также реализующий интерфейс Matrix. Такое решение носит название шаблона проектирования Proxy [5]:

```
public class ProxyMatrix
          implements Matrix {
  private MutableMatrix
                         cache;
  public ProxyMatrix(
                   Matrix a) {
    cache = new
      SquareMatrix(a.size());
    for(int i = 0,
        n = a.size(); i < n;
                         i++) {
    for(int j = 0; j < n;
                         j++) {
        cache.set(i, j,
```

```
                    a.get(i, j));
        }
      }
    }
    public int size() {
      return cache.size();
    }
    public double get(int i,
                      int j) {
      return cache.get(i, j);
    }
}
```

Рассмотренных кэширующих классов может быть несколько, каждый из которых может реализовывать специфический способ кэширования. Например, если мы точно знаем, что результатом вычисления некоторого выражения будет являться матрица специального вида (в частности, матрица, обратная симметричной, тоже всегда симметрична), специальный прокси-класс может хранить результат в классе `SymmetricMatrix` вместо `SquareMatrix`.

Заключение. В нашем исследовании была рассмотрена достаточно простая задача вычислительной алгебры – реализация операций над квадратными матрицами. Однако эта задача является базовой для многих прикладных задач, при решении которых возникает необходимость использовать сложные математические модели. Кроме того, при всей простоте математической постановки подобных задач их реализация на различных вычислительных системах в случае использования больших размерностей становится достаточно сложной задачей, требующей тщательной оптимизации использования различных ресурсов вычислительной системы (процессорное время, объем оперативной и внешней памяти, объем данных, передаваемых по внешним каналам).

При создании программного обеспечения, выполняющего численные расчеты построенных математических моделей, привлекаются специалисты самых разных областей информационных технологий: математики-аналитики, разрабатывающие математические модели; математики-программисты, реализующие логику расчетов построенных моделей; системные программисты, оптимизирующие имеющуюся логику для запуска приложений на спроектированной вычислительной системе. Эффективность работы команды специалистов в таких проектах зачастую обусловлена возможностью разделения задачи на максимально независимые подзадачи.

Предложенная в работе архитектура классов и интерфейсов позволяет выполнять любые операции над матрицами произвольной размерности, а также оптимизировать используемые алгоритмы по различным критериям без необходимости изменения самих этих алгоритмов. Подобный подход позволяет масштабировать такую архитектуру как с целью увеличения производительности вычислительной системы, так и с целью расширения ее функциональных возможностей.

Важной особенностью предложенной архитектуры является ее относительная простота, что достигается за счет использования некоторых широкоизвестных шаблонных решений. Такая простота полученной структуры классов и интерфейсов позволяет знакомить студентов старших курсов специальностей, связанных с информационными технологиями, с подходами, применяемыми в промышленном производстве программного обеспечения.

Также результаты данной работы могут быть адаптированы для использования в различных спецкурсах как по проектированию и разработке программного обеспечения, так и по численным методам решения математических задач, компьютерным сетям и операционным системам, а также математическому моделированию. Подобный подход позволит усилить межпредметные связи и продемонстрировать студентам различные способы разработки программного обеспечения.

ЛИТЕРАТУРА

1. Зарплата в ИТ // dev.by, все о работе в ИТ. – «Дев Бай» [Электронный ресурс]. – 2008–2013. – Режим доступа: http://salaries.dev.by/. – Дата доступа: 29.11.2013.
2. Маркова, Л.В. Объектная реализация методов вычислительной алгебры / Л.В. Маркова, Е.А. Корчевская, А.Н. Красоткина // Весн. Віцебск. дзярж. ун-та. – 2013. – № 2(74). – С. 18–22.
3. Маркова, Л.В. Вычислительные методы алгебры. Практикум: пособие / Л.В. Маркова, Е.А. Корчевская, А.Н. Красоткина. – Витебск: ВГУ имени П.М. Машерова, 2013. – 148 с.
4. Мартин, Р.С. Быстрая разработка программ. Принципы, примеры, практика / Р.С. Мартин, Д.В. Ньюкирк, Р.С. Косс. – Киев: Вильямс, 2004. – 752 с.
5. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. – СПб.: Питер, 2010. – 368 с.
6. Таненбаум, Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. – СПб.: Питер, 2003. – 877 с.

Поступила в редакцию 05.11.2013. Принята в печать 19.11.2013
Адрес для корреспонденции: e-mail: yermochenko@tut.by – Ермоченко С.А.