не d-разделены относительно C, говорят, что X и Y d-связаны относительно C. Определение d-разделения двух узлов может легко быть распространено на d-разделение двух непересекающихся множеств узлов составляющих путь в графе [7, 8].

То, что для разделения, объединения и добавления связей используется одна и та же мера позволяет сравнивать пользу от этих операций друг с другом и, соответственно трудоемкость при пересчете.

Достоинством описанного алгоритма тестирования работы сети является его гибкость. Алгоритм допускает исключение некоторого множества путей пересчета из списка с целью повышения точности других путей пересчета.

Литература

1. Jie Cheng. David Bell, Weiru Liu, Learning Bayesian Networks from Data: An Efficient Approach Based on Information Theory UCLA, 1998
2. Witten, I. H. (Ian H.). —Data Mining. Practical Machine Learning Tools and Techniques
3. Han J., Kamber M. Data Mining: Concepts and Techniques. — Morgan Kaufmann Publishers, 2000.
4. С. Хабаров. - Конспект лекций. 2003.
5. Hand D. J., Mannila H., Smyth P. Principles of Data Mining. — MIT Press, 2000.
6. Dempster A. P., Laird N. M., Rubin D. B. Maximum likelihood from incomplete data via the EM algorithm // J. of the Royal Statistical Society, Series B.
7. Learning Bayesian Network Structure from Distributed Data, R. Chen K. Sivakumar, 2002
8. Pearl, J., Probabilistic reasoning in intelligent systems: networks of plausible inference, Morgan Kaufmann, 1988
9. Искусственный интеллект, современный подход. Стюарт Рассел, Питер Норвиг. М. 2006

РЕАЛИЗАЦИЯ РОЛЕЙ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ СО СТАТИЧЕСКОЙ ТИПИЗАЦИЕЙ

А. П. Побегайло
Беларусь, г. Минск

Введение

Прогресс в развитии отрасли программного обеспечения привел к разработке все более совершенных и технически сложных программных систем. Реализация таких систем требует глубокого понимания и точной формализации процессов, которые автоматизирует или которыми управляет программная система. Очевидно, что разработка таких систем невозможна без их моделирования. В связи с этим все более актуальным становится онтологический подход к проектированию программных систем. Как правило, при таком подходе к моделированию системы требуется определиться со способом реализацией ролей.

В данном докладе рассмотрен вопрос реализации ролей в языках программирования со статической типизацией. В качестве языка программирования, который выбран для демонстрации реализации, используется язык программирования C++, так как он стандартизирован и широко используется при разработке программных систем.

1. Онтологический подход к проектированию программных систем

В настоящее время при проектировании программных систем требуется разрабатывать довольно сложные протоколы и интерфейсы. Это требует точного определения абстрактных понятий из предметной области. То есть присутствует некоторый философский подход к проектированию системы. Используя философские термины, которые, впрочем, сейчас также широко используются и в исследованиях по моделированию систем, можно сказать, что процесс разработки программных систем есть комбинация онтологического и эпистемологического подходов [1, 2]. Или другими словами модель системы есть синтез классов, описывающих абстрактные и реальные объекты.

Онтологический подход к разработке программной системы подразумевает построение концептуальной модели этой системы. Термин онтология означает систему концепций и отношений между ними, причем определение этой системы носит декларативный характер. Каждая концепция рассматривается как сущность, которая обладает свойствами и связями с другими концепциями. Это очень напоминает разработку интерфейсов или абстрактных классов, содержащих только чисто виртуальные функции. Однако заметим, что концепция не подразумевает способ её реализации.

С другой стороны, проблематично, построить реальную систему, которая описывается только концепциями или интерфейсами. Как правило, при проектировании системы также прорабатываются реальные или, используя философский термин, материальные классы, которые не содержат виртуальных функций. Можно сказать, проработка реальных классов требует применения эпистемологического подхода к моделированию системы.

2. Определение роли

Как следует из предыдущего изложения, при проектировании программной системы возникают абстрактные концепции и конкретные классы. Не всегда конкретные классы используются только как типы параметров или значений, возвращаемых функциями членами абстрактных классов. Часто встречаются такие ситуации, в которых конкретные классы реализуют концепции. Вопрос реализации конкретной концепции требует отдельного рассмотрения. Если конкретный класс используется каким-либо образом для реализации концепции, то можно сказать, что он исполняет роль этой концепции.

Часто, абстрагируясь от вопросов проектирования систем, при определении ролей используют следующий подход. Все классы разбивают на две категории: естественные классы (natural or intrinsic classes) и ролевые классы (role classes) или просто роли. Подразумевается, что естественные классы имеют конкретные свойства и отношения с другими естественными классами. С дру-

гой стороны, роли исполняются естественными классами, а сами по себе не имеют конкретных свойств и отношений. Например, человек может играть роль врача, учителя и так далее. Причем, часто возникает ситуация, когда один конкретный класс может играть несколько ролей.

Отсюда видно, что при реализации ролей должна решаться проблема повторного использования кода, так как реализация ролей должна содержать реализацию конкретных классов, играющих эту роль. Возможные подходы к реализации ролей рассмотрены в следующих разделах.

3. Возможные способы реализации ролей

В языках программирования со статической типизацией существует три подхода к реализации ролей, преимущества и недостатки каждого из которых кратко рассмотрены ниже [3, 4].

Первый подход к реализации ролей заключается в применении композиции классов. То есть объект ролевого класс может содержать объект или указатель на объект естественного класса, которому переадресует вызовы методов, семантически связанных с естественным классом. Данный подход довольно универсален и, наверное, почти всегда применим. Единственным его недостатком являются дополнительные затраты на косвенные вызовы методов вложенного объекта естественного класса.

Второй подход к реализации ролей заключается в использовании наследования классов. При этом возможно как наследование ролей от естественного класса, так и наоборот. Однако, такой подход возможен только в случае, если интерфейс естественного класса соответствует интерфейсу ролевых классов. Как будет показано ниже, это не всегда так.

Третий подход к реализации ролей заключается в использовании специальных языковых средств. Но такие средства еще недостаточно проработаны и пока редко используются в зыках программирования.

4. Реализация при помощи шаблонов

В системах компьютерной графики и геометрии приходится часто работать с различными геометрическими объектами. Как правило, для описания этих объектов используются точки, плоскости, векторы или декартовые координаты. Как соотносятся между собой эти элементарные классы? Есть ли между ними наследование? Решение этих вопросов определяет способ их программирования и, в конечном счете, определяет интерфейс системы, так как эти объекты являются базовыми.

Рассмотрим интерфейс класса декартовых координат. Для простоты будем считать, что проектируется система, работающая с геометрическими объектами на плоскости.

```
class Coord
{
 double x, y;
public:
 // constructors
 Coord() {}
```

```
  Coord(const double& _x, const double& _y): x(_x), y(_y) {}
  // member operators
  Coord& operator +=(const Coord&);
  Coord& operator -=(const Coord&);
  Coord& operator *=(const double& d);
  // friend functions
  friend double x(Coord& c) { return c.x; }
  friend double y(Coord& c) { return c.y; }
  // friend opeartors
  friend Coord operator +(const Coord&, const Coord&);
  friend Coord operator -(const Coord&, const Coord&);
  friend Coord operator *(const double&, const Coord&);
  friend Coord operator *(const Coord&, const double&);
};
```

Совершенно аналогичный интерфейс имеют классы векторов и ковекторов

```
class Vector;
class Covector;
```

или, допустим, классы векторов строк и векторов столбцов, нужно только изменить в интерфейсе класса декартовых координат имя класса. Очевидно, что декартовые координаты это реальный класс, а векторы и ковекторы это абстрактные объекты, которые описываются координатами. Поэтому, декартовые координаты играют роли векторов и ковекторов. Естественно возникает вопрос, как реализовать эти роли.

Можно использовать композицию классов, то есть определить внутри классов Vector и Covector объект типа декартовых координат и переадресовывать все операции ему. Но такие косвенные вызовы сильно замедлят работу системы, так геометрические объекты, как правило, описываются большими массивами элементарных геометрических объектов.

Наследование в этом случае также не работает, так как интерфейсы ролевых классов отличаются от интерфейса естественного класса. Это различие можно обойти, используя роли как наследники естественного класса и определяя в них конструкторы для преобразования к базовому типу, например,

```
Vector::Vector(const Coord& c): x(c.x), y(c.y) {}
Covector::Covector(const Coord& c): x(c.x), y(c.y) {}
```

Тогда в ролевых классах интерфейс базового класса можно не дублировать. Но здесь возникает следующая проблема. В этом случае операторы могут работать над векторами разных типов и результат работы такой операции также может быть присвоен любому вектору. Это противоречит семантике этих операций, например, мы не можем складывать вектор столбец с вектором строкой.

Поэтому приходится просто дублировать код для классов векторов и ковекторов. Для автоматизации этого дублирования можно использовать шаблоны. Тогда интерфейс шаблона класса декартовых координат будет выглядеть следующим образом:

```
template<class Role> class Coord
{
 double x, y;

public:
 // constructors
 Coord<Role>() {}
 Coord<Role>(const double& _x, const double& _y): x(_x), y(_y) {}
 // member operators
 Coord<Role>& operator +=(const Coord<Role>&);
 Coord<Role>& operator -=(const Coord<Role>&);
 Coord<Role>& operator *=(const double& d);
 // friend functions
 friend double x(Coord<Role>& r) { return r.x; }
 friend double y(Coord<Role>& r) { return r.y; }
 // friend opeartors
 friend Coord<Role> operator +(const Coord<Role>&,
                               const Coord<Role>&);
 friend Coord<Role> operator -(const Coord<Role>&,
                               const Coord<Role>&);
 friend Coord<Role> operator *(const double&, const Coord<Role>&);
 friend Coord<Role> operator *(const Coord<Role>&, const double&);
};
```

Теперь можно определять любой класс, роль которого играют декартовые координаты. В нашем случае классы векторов и ковекторов определяются так

```
class vector;
class covector;
```

а работать с объектами этих классов можно обычным образом:

```
Coord<vector> a(1, 1), b(2, 2), c;
a += b;
c = a + b;

Coord<covector> u(1, 1), v(2, 2), w;
u += v;
w = u + v,
```

Если при реализации роли необходимо, чтобы инициализация естественного класса выполнялась динамически на стороне сервера, то в этом случае необходимо использовать композицию классов, причем вложенный объект естественного класса определять через указатель.

Сходные проблемы возникают при использовании матриц для моделирования геометрических и физических объектов.

Заключение

В предложенном докладе рассмотрены варианты реализации ролей в языках программирования со статической типизацией. Показано, что не всегда удается избежать дублирования кода при программировании ролей.

Предложено решение этой проблемы с использованием шаблонов классов и функций.