Gem #8: Фабричные функции

Автор: Matthew Heaney (On2 Technologies)

Краткое содержание: Gem Ada #8 – Фабричные функции позволяют создавать объекты какого-либо класса , когда известен только объект другого класса. С их помощью можно реализовывать присваивание объектам надклассовых типов, чтобы избежать исключений, вызванных несоответствием тегов.

Давайте начнем…

Предположим, что у нас есть универсальный пакет, который объявляет класс стека. Корень иерархии был бы следующие:

```
generic
  type Element_Type is private;
package Stacks is
  type Stack is abstract tagged null record;

  procedure Push
   (Container : in out Stack;
    Item      : in     Element_Type) is abstract;
 …
end Stacks;
```

Предположим, что в классе существуют различные конкретные типы, например неограниченный стек (который автоматически растет по мере необходимости) и ограниченный стек (реализованный как массив фиксированного размера).

Теперь предположим, что мы хотим назначить один стек другому, независимо от конкретного типа стека, примерно так:

```
procedure Op (T : in out Stack'Class; S : Stack'Class) is
begin
   T := S;  -- raises exception if tags don't match
   …
end;
```

Этот текст программы компилируется, но во время выполнения может быть сгенерировано исключение, как только тег целевого стека не совпадет тегом исходного стека. Наша цель здесь состоит в том, чтобы выяснить, как присвоить объекты стека (чей тип всего класса) способом, таким образом, что присвоение будет работать, гарантированно не генерировать исключение из-за несоответствия тега стека.

Один из способов сделать это – создать операцию стиля присваивания примитивной для этого типа, чтобы выполнялась диспетчеризация согласно типу целевого стека. Если тип исходного стека является классом, то не может быть несоответствие тега (и, следовательно, исключение), поскольку есть только один управляющий параметр. (Обратите внимание на то, что Вы могли сделать это по другому, выполнять диспетчеризирование на теге исходного стека. Вы могли даже сделать работу всего класса, так, чтобы она не должна была диспетчеризироваться вообще. Идея состоит в том, чтобы избежать передавать больше, чем единственный управляемый операнд.)

Процедура Assign может быть объявлена как это:

```
procedure Assign
   (Target : in out Stack;
    Source : Stack'Class) is abstract;
```

который позволил бы нам переписывать вышеупомянутый оператор присваивания как:

```
procedure Op (T : in out Stack'Class; S : Stack'Class) is
begin
   T.Assign (S);  -- dispatches according T's tag
   …
end;
```

Каждому типу в классе придется переопределить Assign. В качестве примера давайте рассмотрим шаги, необходимые для реализации операции для типа ограниченного стека. Его спецификация будет выглядеть так:

```
generic
package Stacks.Bounded_G is
   type Stack (Capacity : Natural) is
     new Stacks.Stack with private;

   procedure Assign
     (Target : in out Stack;
      Source : Stacks.Stack'Class);
   …
private
   type Stack (Capacity : Natural) is
     new Stacks.Stack with
   record
      Elements  : Element_Array (1 .. Capacity);
      Top_Index : Natural := 0;
   end record;
end Stacks.Bounded_G;
```

Это всего лишь каноническая реализация ограниченной формы контейнера, которая использует дискриминацию для управления количеством хранения для объекта. Интересной частью является реализация операции Assign, поскольку нам нужен способ перебора элементов в стеке источника. Вот скелет реализации:

```
procedure Assign
   (Target : in out Stack;  -- bounded form
    Source : Stacks.Stack'Class)
is
   …
begin
    …
   for I in reverse 1 .. Source.Length loop
       Target.Elements (I) := <get curr elem of source>
       <move to next elem of source>
    end loop;
    …
end Assign;
```

Обратите внимание на то, что, предполагая, что мы посещаем элементы исходного стека в порядке сверху вниз, дело не в том, чтобы выталкивать элементы в целевой стек, так как если бы мы сделали это, то элементы оказались бы в обратном порядке. Вот почему мы заполняем массив целевых стеков в обратном порядке, начиная с самого большого индекса (вершина стека) и работая назад (в сторону нижней части стека).

Вопрос в том, как вы перебираете исходный стек? Предположим, что каждый конкретный тип в классе стека имеет свой собственный тип итератора, соответствующий конкретному представлению этих стеков (подобно тому, как реализованы контейнеры в стандартной библиотеке). Проблема заключается в том, что тип формального параметра исходного стека является общедоступным. Как получить итератор для фактического параметра исходного стека, если его конкретный тип неизвестен (то есть неизвестен статически)?

Ответ, просто запросите у стека для каждого элемента! Тег-тип имеет операции отправки, некоторые из которых могут быть функциями, поэтому здесь нам просто нужна диспетчерская функция для возврата объекта итератора. Идиома отправки объекта, тип которого находится в одном классе, возвращает объект, тип которого находится в другом классе, называется «фабричной функцией» или «диспетчером-конструктором».

Операция может быть примитивной только для одного помеченного типа, поэтому, если операция отправляется по параметру стека, возвращаемый тип функции должен быть общедоступным. Теперь мы вводим тип Cursor, корень иерархии итератора стека и изменяем класс стека с «фабричной функцией» для курсоров:

```
type Cursor is abstract tagged null record;  -- the iterator

function Top_Cursor  -- the factory function
  (Container : not null access constant Stack)
  return Cursor'Class is abstract;

… -- primitive ops for the Cursor class
```

Каждый тип класса стека переопределяет Top_Cursor, чтобы вернуть курсор, который можно использовать для доступа к элементам этого объекта стека. Теперь мы можем завершить выполнение операции Assign для ограниченных стеков следующим образом:

```
procedure Assign
  (Target : in out Stack;
   Source : Stacks.Stack'Class)
is
   C : Stacks.Cursor'Class := Source.Top_Cursor;  -- dispatches

begin
   Target.Clear;

   for I in reverse 1 .. Source.Length loop
      Target.Elements (I) := C.Element;  -- dispatches
      C.Next;  -- dispatches
   end loop;

   Target.Top_Index := Source.Length;
end Assign;
```

Параметр Source имеет тип класса, что означает вызов диспетчеров Top_Cursor (поскольку Top_Cursor является примитивным для этого типа). Это именно то, что мы хотим, так как разные типы стеков будут иметь разные представления и поэтому потребуются разные типы курсоров. Объект cursor (здесь, C), возвращаемый фабричной функцией, сам по себе является классом, что означает, что операции курсора также вызывает диспетчер. Вызов функции C.Element возвращает элемент Source в текущей позиции курсора, а C.Next перемещает курсор в следующую позицию (в сторону нижней части стека).