УДК 004.435

# ПРИСОЕДИНЯЕМЫЕ ТИПАЖИ В JAVA: РАСШИРЕНИЕ ЯЗЫКА И ОБЛАСТЬ ПРИМЕНЕНИЯ

**А. А. Седунов**

*Воронежский государственный университет*

Поступила в редакцию 01.03.2010 г.

**Аннотация.** В данной статье представлен один из вариантов расширения языка Java, основанный на механизме присоединяемых типажей. Описанное расширение позволяет использовать на платформе Java гибкую форму наследования, при которой функциональность класса и его связи наследования могут меняться в зависимости от контекста. Рассмотрены необходимые изменения в синтаксисе, системе типов Java, ключевые аспекты семантики присоединяемых типажей, а также основные области их применения.
**Ключевые слова:** ООП, тип, типаж, модульность.

**Abstract.** This article describes one of possible Java language extensions based on attachable trait approach. The extension allows using flexible form of inheritance on the Java platform so that class behavior and its inheritance links can change depending on the context. Necessary syntax changes, type system extension as well as key aspects and major application areas of attachable trait semantics are presented.
**Key words:** OOP, type, trait, modularity.

## ВВЕДЕНИЕ

Одна из проблем, известных в практике объектно-ориентированной разработки программного обеспечения (т. н. проблема "дополнительного метода" (augmenting method problem) [1]), состоит в том, что традиционная реализация ООП в основных языках программирования затрудняет свободное расширение существующей иерархии классов путем введения дополнительных операций. При этом существенно ограничивается модульность программной системы, поскольку расширение функциональности ее компонентов невозможно без модификации существующего программного кода.

В данной работе мы предлагаем в качестве решения расширение традиционной парадигмы ООП с помощью механизма присоединяемых типажей. Понятие *типажа* (trait) в контексте ООП является обобщением интерфейса, используемых в некоторых языках, например, Scala [4], Squeak, Perl 6 и др. Типаж представляет собой совокупность методов, обладающих логической связностью и характеризующих определенный аспект поведения. В отличие от интерфейсов, методы типажей не обязательно должны быть абстрактными и могут содержать элементы реализации. Основное назначение типажей состоит в добавлении связанного с ними поведения к существующим классам путем наследования, либо при создании их экземпляров. Идея разработанного нами подхода состоит в развитии типажей и дополнении объектно-ориентированного языка "гибкой" формой наследования на их основе, позволяющей модифицировать связи наследования между классами в зависимости от контекста.

## 1. ОПИСАНИЕ РАСШИРЕНИЯ ЯЗЫКА

Функциональность, вводимая предлагаемым расширением языка Java, сводится к следующему:

— структура описания типажа;
— операции присоединения типажа;
— неявный импорт типажей.

В настоящей статье мы не приводим строго описания расширения языка, концентрируясь, главным образом, на неформальном описании структуры и, частично, семантики присоединяемых типажей, а также вариантов их использования в ряде задач. В дальнейших работах мы более подробно остановимся на формальных аспектах предлагаемого расширения Java.

Определение типажей

Синтаксис описания типажа имеет следующий вид:

```
TraitDecl ::=
Modifiers? "trait" ID TypeParameters?
ExtendsDecl? ContextDecl ClassBody
ContextDecl ::= "for" Type
```

Для пояснения структуры этого определения мы проведем сравнение с описанием Java-интерфейса. Сходство с интерфейсами состоит в следующем:

— Все члены типажа автоматически получают уровень доступа `public` (модификатор `public` также может присутствовать явно).

— Все поля и вложенные типы (классы/интерфейсы/типажи) автоматически являются статическими, а поля также и финальными (модификаторы `static` и `final` могут присутствовать явно).

— Типажи поддерживают множественное наследование (типаж может расширять несколько типажей).

— Типаж не может наследовать класс.

В контексте рассматриваемого расширения Java-интерфейсы считаются частным случаем типажей, в которых все методы являются абстрактными. В частности, типажи могут расширять Java-интерфейсы (однако обратное свойство не выполняется).

В отличие от интерфейсов, типажи предоставляют следующие возможности:

— Методы типажей, как и методы классов, могут сопровождаться реализацией. Если тело метода опущено, то он автоматически считается абстрактным (модификатор `abstract` также может присутствовать в явном виде). Методы типажей могут реализовывать и перекрывать унаследованные методы, для них, как и для методов класса, допустим модификатор `final`.

— Типаж автоматически считается абстрактным, если абстрактен хотя бы один его собственный или унаследованный метод (модификатор `abstract` может также присутствовать явно).

— Типаж может содержать определение контекста в части `for`. Контекст описывает тип, методы которого типаж может использовать в своей реализации и к которому он может быть динамически присоединен одной из специальных операций (`with` или `attached`).

При множественном наследовании типажей возможна ситуация, в которой два или более родительских типажа содержат методы с одинаковыми сигнатурами. Так как определяемый типаж не имеет возможности различить эти методы, возникает неоднозначность. Для разрешения неоднозначности мы вводим следующие правила. Пусть `X` — определяемый типаж, `A` и `B` — два его родительских типажа, `mA` и `mB` — два метода с одинаковыми сигнатурами, определенные соответственно в `A` и `B`.

1. Если оба метода `mA` и `mB` абстрактные, то типаж `X` наследует тот метод, который является перекрывающим (при этом используется правило выбора перекрывающего метода для множественного наследования интерфейсов).

2. Если один из методов (скажем, `mA`) абстрактный, а другой (`mB`) — конкретный, то типаж `X` наследует конкретный метод `mB` при условии, что он является корректной реализацией метода `mA` (в противном случае определение типажа `X` считается некорректным).

3. Если оба метода `mA` и `mB` являются конкретными, то определение типажа `X` считается некорректным.

Данная стратегия разрешения неоднозначности представляет собой адаптированный вариант правил построения типов-пересечений из ранее разработанного нами языка спецификаций JLS [5]. Заметим, что приведенные правила (в первую очередь, правило запрета комбинирования двух неабстрактных методов) носят достаточно консервативный характер. В дальнейшем мы планируем разработку более гибкого механизма, позволяющего идентифицировать методы, унаследованные от разных родителей (один из вариантов, реализованный также в языке JLS, состоит в применении т. н. производных типов, позволяющих переименовывать методы).

В разделе "Варианты использования" мы рассмотрим конкретные примеры описания и использования типажей.

Внутри описания типажа T с явным указанием контекста C ссылка this имеет фактический тип [C with T] (см. далее описание комбинированных типов и семантики вызова их методов).

## ВНЕШНИЕ МЕТОДЫ

Так как во многих случаях присоединяемые типажи содержат только один метод, мы приняли решение ввести специальный синтаксис для описания таких типажей, называемых далее внешними методами, а также расширить систему типов Java функциональными типам.

Синтаксис функционального типа имеет вид:

```
FuncType ::= "[" ArgTypes "=>" Type "]"
ArgTypes ::= Type | ArgTypes "," Type
```

Перед символом => перечисляются типы аргументов, а тип, находящийся после => характеризует возвращаемое значение. В данном случае мы не используем описание аналогичных типов, известных в функциональных языках (с последовательной передачей аргументов, или каррированием), так как оно не соответствует семантике передачи параметров в Java. Например, тип `[Object => void]` описывает функцию без возвращаемого значения с параметром типа `Object`, тип `[int, String => String]` — функция с результатом типа `String` и двумя параметрами, имеющими типы `int` и `String` соответственно и т. д.

Правила определения подтипов определяются обычным для функциональных типов образом (с поправкой на отсутствие каррирования): `[S1, …,Sn => S]` является подтипом `[T1, …,Tm => T]`, если выпоняются следующие условия:

1. `n = m`;
2. `Ti` — подтип `Si` для всех `1<= i <= n`;
3. `S` — подтип `T`.

Например, `[Object, String => void]` является подтипом `[String, String => void]`.

Фактически каждому функциональному типу `[S1,…Sn => S]` соответствует автоматически генерируемый абстрактный типаж следующего вида:

```
trait Func {
   public S apply(S1 p1, …, Sn pn);
}
```

Так, тип `[Object, String => void]` представляет собой типаж с методом

public void apply(Object p1, String p2);

Имя этого типажа недоступно в программе явным образом (идентификатор `Func` выбран произвольным образом для соблюдения формального синтаксиса). Тем не менее, в остальном он подчиняется тем же правилам, что и остальные типажи — в частности, он может выступать в качестве родительского типажа. Эта возможность непосредственно используется для определения внешних методов.

```
ExternalMethodDecl ::=
Modifiers? "method"
MethodHeader ExtendsDecl? Context-
Decl?
MethodBody?
```

Синтаксически определение внешнего метода представляет собой описание обычного метода с добавлением ключевого слова `method` и модификаторов, а также опциональных секций наследования (`extends`) и контекста (`for`). Семантика внешнего метода реализуется путем его трансформации в фактическое определение типажа. Имя типажа является производным от имени метода (совпадает с ним, за исключением того, что первый символ переводится в верхний регистр). Именно определение вида

```
method S methodName (S1 p1,…, Sn
pn) {…}
```

раскрывается следующим образом:

```
trait MethodName extends [S1, …,
Sn => S] {
   public final S apply(S1 p1, …,
Sn pn) {
      return methodName(p1, …, pn);
}

public S methodName(S1 p1, …, Sn
pn) {…}
}
```

Модификаторы, записанные перед словом `method`, переносятся на описание типажа. Типы, указанные в секции `extends` (если она присутствует), добавляются к секции `extends` типажа. Аналогичное правило касается секции описания контекста `for`. Если тело метода отсутствует, сгенерированный типаж также будет иметь метод без тела (т. е. абстрактный).

Пример описания внешнего метода и соответствующего ему типажа приводится в следующем разделе.

Заметим, что внешний метод не может генерировать проверяемые исключения, так как он используется для реализации метод `apply()` в соответствующем типаже, а последний описан в типажах функциональных методов без секции `throws`. В дальнейшем мы планируем расширить синтаксис функциональных типов так, чтобы сделать возможным включение информации о потенциальных исключениях.

## КОМБИНИРОВАННЫЕ ТИПЫ. ВЫЗОВ МЕТОДОВ ТИПАЖА

Для описания операций присоединения типажей мы вводим в язык еще одну группу типов, далее называемых комбинированными. Эти типы не доступны программе явным образом — они автоматически конструируются

компилятором в зависимости от контекста, определяемого одной из операций присоединения. Мы будем использовать для них обозначение `[S with T1, …, Tn]`, где `S` — произвольный ссылочный тип (называемый ядром), а `T1, …, Tn` — типажи.

Комбинированный тип обладает тем же набором полей и вложенных типов, что и его ядро. Набор методов комбинированного типа определяется как объединение методов его ядра и типажей, за тем исключением, что если в ядре и типаже (или двух типажах) описаны методы с одинаковыми сигнатурами, метод типажа, стоящего на большей позиции поглощает другой метод. В частности, метод типажа всегда поглощает метод ядра с такой же сигнатурой, а поглощение методов между типажами зависит от их расположения в описании комбинированного типа. В связи с этим, например, записи [S with T1, T2] и [S with T2, T1] обозначают разные типы.

Вызов метода комбинированного типа во время выполнения программы происходит следующим образом. Если метод принадлежит ядру, то вызов происходит по правилам исходного языка Java. Если же метод принадлежит типажу T, то выполняется следующая процедура:

1. Определить множество X всех типажей, которые прямо или косвенно наследуют T и контекст которых является супертипом ядра (заметим, что это исключает типажи, в которых отсутствует описание контекста).

2. Пока в множестве X есть два таких типажа, что контекст одного является подтипом другого, удалить второй типаж из X.

3. Пока в множестве X есть два таких типажа, что один из них является подтипом другого, удалить второй типаж из X.

4. Возможны 3 случая:

a. множество X пусто — в этом случае генерируется исключение UnresolvedTraitMethodException (не удалось не найти метод, удовлетворяющий условиям вызова);

b. множество X содержит единственный элемент, типаж T0 — в этом случае выполняется вызов метода T0 с сигнатурой, соответствующей условиям вызова;

c. множество X содержит более одного элемента — в этом случае генерируется исключение AmbiguousTraitMethodException (возникла неоднозначность при вызове метода).

Текущая реализация присоединяемых типажей не гарантирует существование и однозначность метода типажа, вызываемого через комбинированный тип — поэтому в случае ошибки генерируется соответствующий объект-исключение. Заметим, однако, что существует возможность представления подобной гарантии на этапе статического анализа программы (подобно тому, как Java контролирует вызов методов абстрактных классов) — необходимые для этого ограничения системы типов с типажами являются предметом текущего исследования.

## ПРИСОЕДИНЕНИЕ ТИПАЖЕЙ К ОБЪЕКТАМ И ТИПАМ

Использование типажей реализуется в два этапа: на первом этапе выполняется присоединение типажа к целевому типу или объекту, на втором происходит вызов метода типажа.

Вначале рассмотрим способы присоединения типажа. Представленный вариант реализации типажей предполагает 3 варианта присоединения:

— присоединение на уровне объекта;

— блочное присоединение;

— произвольное присоединение.

Присоединение на уровне объекта представляет собой операцию и имеет следующий синтаксис:

WithExpression ::= “(“ Expression “with” Type “)”

Выражение, стоящее перед ключевым словом `with` должно иметь ссылочный тип — оно представляет собой объект, к которому присоединяется типаж, указанный после `with`. Тип `with`-выражения определяется следующим образом. Пусть `T` — тип выражения, а `T0` — присоединяемый типаж, тогда

— если `T` — комбинированный тип `[S with T1, …, Tn]`, то `with`-выражение имеет тип `[S with T1, …, Tn, T0]`;

— в противном случае `with`-выражение имеет тип `[T with T0]`.

Блочное присоединение представляет собой оператор следующего вида:

AttachedStatement ::= “attached” Type “to” Type CompoundStatement

Здесь присоединяемый типаж указывается перед словом `to`, а базовый тип, к которому выполняется присоединение — после `to`. Эффект от оператора `attached T0 to T {…}` состоит в том, что внутри блока `{…}`

— все выражения `e`, тип которых является подтипом `T`, автоматически заменяются на `(e with T0)`;

— для любого вхождения типа T′, являющегося подтипом T, комбинированные типы [T′ with T1, …, Tn] заменяются на [T′ with T1, …, Tn, T0];

— для любого вхождения типа T′, являющегося подтипом T, но не являющегося ядром комбинированного типа, T′ заменяется на [T′ with T0].

Произвольное присоединение представлено двумя объявлениями:

```
AttachDecl ::= "attach" Type "to" Type
DetachDecl ::= "detach" Type "from" Type
```

Эти объявления должны находиться на одном лексическом уровне, т. е.

— вне определения класса, но в одном исходном файле;

— вне определения метода, но внутри одного классе;

— внутри одного метода.

Кроме того, если на одном из указанных уровней встречается объявление attach (detach), то на этом же уровне после него (перед ним) должно находиться соответствующее объявление detach (attach). Соответствующими считаются объявления вида attach T0 to T и detach T0 from T. Другие комбинации attach и detach считаются некорректными.

Эффект от произвольного присоединения аналогичен эффекту оператора attached, но распространяется на фрагмент программного кода, расположенный между attach и ближайшим к нему соответствующим объявлением detach. Таким образом, данная форма позволяет выполнять присоединение в нескольких методах или классах (при условии, что они описаны в одном файле).

## 2. ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ

Далее мы рассмотрим основные варианты использования присоединяемых типажей в контексте объектно-ориентированного подхода, а также решение с их помощью проблем, обозначенных в предыдущем разделе.

Расширение функциональности классов

Присоединяемые типажи могут использоваться для расширения функциональности классов (интерфейсов) без необходимости модификации их программного кода, что особенно важно в тех случаях, когда исходные тексты классы недоступны или разработка приложения ведется коллективно.

В качестве примера рассмотрим расширение интерфейса Iterator операцией forEach(), позволяющей выполнить заданную функцию для каждого элемента итератора. Напомним определение Iterator (в этом примере мы для простоты игнорируем параметрический полиморфизм Java и запишем интерфейс без типов-параметров):

```
interface Iterator {
   public boolean hasNext();
   public Object next();
   public void remove();
}
```

Теперь опишем типаж, реализующий операцию forEach() и типаж, описывающий унарную функцию:

```
trait ForEachIterator for Iterator {
   public void forEach([Object =>
void] f) {
          while (hasNext()) {
                f(next());
          }
    }
}
```

После этого для использования описанной функциональности достаточно обратиться к итератору в контексте, присоединяющем типаж ForEachIterator к интерфейсу Iterator, например, с помощью оператора attached:

```
Set mySet = new HashSet();
...
attached ForEachIterator to It-
erator {
    StringBuilder sb = new
StringBuilder();

    sb.append("{ ");
mySet.iterator().forEach(
         [Object val => void] {
              sb.append(val).
append(" ");
         }
);
    sb.append("}");
}
```

Таким образом, в данном примере мы получаем возможность раздельного описания 3-х компонентов программной логики: генерация последовательности (Iterator), алгоритм обхода (ForEachIterator) и алгоритм обработки элементов ([Object => void]).

Заметим также, что типаж ForEachIterator содержит только один метод, поэтому его определение можно было реализовать с помощью конструкции method (при этом, конечно, имя типажа, используемого в операции with должно быть ForEach):

```
method void forEach([Object => void] f) for Iterator {
    while (hasNext()) {
f(next());
    }
}
```

Как уже говорилось ранее, подобное описание фактически разворачивается в определение типажа, производного от Method:

```
trait ForEach extends [[Object => void] => void] for Iterator {
    public final void invoke([Object => void] f) {
        forEach(f);
}

public void forEach([Object => void] f) {
        while (hasNext()) {
            f(next());
        }
}
}
```

Так как присоединение типажей, строго говоря, не является наследованием, оно может применяться для закрытых классов (отмеченных модификатором final). В следующем примере показано, как типаж Romanizer (реализующий преобразование целого числа в римскую запись) может быть применен к объекту закрытого типа Integer:

```
trait Romanizer for Number {
public String toRomanString() {
    ...
}
}
...

Integer i = 123;
String s1 = i.toString();
    // "123"
String s2 = (i with Romanizer).toRomanString(); // "CXXIII"
```

Таким образом, присоединяемые типажи позволяют расширять поведение существующих классов, приписывая им методы, определенные разработчиком. Это, в частности, позволяет решить проблему соотношения т. н. "кратких" и "детализированных" интерфейсов (sparse/rich interface) [].

Обход структур данных. Паттерн Visitor

Использование присоединяемых типажей также позволяет упростить реализацию паттерна Visitor [], в частности, в задачах, требующих реализации той или иной формы обхода сложной структуры данных.

Рассмотрим следующую структуру классов, описывающих арифметические выражения (без переменных):

```
abstract class Expr  {
    public abstract int eval();
}

class Const extends Expr {
    public final int value;

    public Const(int value) {
        this.value = value;
    }

public final int eval() {
    return value;
}
}

class Sum extends Expr {
    public final Expr left, right;

    public Sum(Expr left, Expr right) {
        this.right = left;
        this.right = right;
    }

    public final int eval() {
    return left + right;
}
}

class Product extends Expr {
    public final Expr left, right;

    public Product(Expr left, Expr right) {
        this.right = left;
        this.right = right;
    }
```

```
    public final int eval() {
    return left*right;
  }
  }
```

Для простоты примера мы игнорируем уровни доступа и используем неизменяемые поля. В данном случае каждый класс описывает данные соответствующей разновидности выражений и, кроме того, характеризуется аспектом поведения, прототип которого описан в базовом классе `Expr` — операция `eval()` для вычисления значения выражения.

Предположим, что нам требуется реализовать обход структуры выражения с построением его текстового представления. Включать метод преобразования (например, `convert()`) непосредственно в классы `Expr` нецелесообразно, так как на практике может требоваться более одной реализации процедуры обхода (например, для инфиксной и постфиксной записи). Кроме того, код классов-выражений может быть недоступен для изменения (например, они могут быть частью внешней библиотеки). Вынесем метод `convert()` в отдельный интерфейс `Convertible`:

```
  trait Convertible {
    public abstract void
apply(StringBuilder sb);
  }
```

Для того, чтобы реализовать конкретный вариант обхода (например, инфиксную запись), построим для каждого класса-выражения соответствующий ему типаж, реализующий интерфейс `Convertible`:

```
  trait InfixExprCollector extends
Convertible depends on Expr {
    public void apply(StringBuilder
sb) {
          sb.append("?");
  }
  }

  trait InfixConstCollector
  extends InfixStringCollector depends
on Const {
  public void apply(StringBuilder
sb) {
    sb.append(value);
  }
  }

  trait InfixSumCollector
  extends InfixStringCollector depends
on Sum {
  public void apply(StringBuilder
sb) {
  sb.append('(');
  (left with InfixExprCollector).
apply(sb);
  sb.append(' + ');
  (right with InfixExprCollector).
apply(sb);
  sb.append(')');
  }
  }

  trait InfixProductCollector
  extends InfixStringCollector depends
on Product {
  public void apply(StringBuilder
sb) {
  (left with InfixExprCollector).
apply(sb);
  sb.append('*');
  (right with InfixExprCollector).
apply(sb);
  }
  }
```

Мы можем применить описанный алгоритм обхода к переменной типа `Expr`, воспользовавшись присоединением базового типажа `InfixExprCollector`:

```
  Expr expr = new Product(
  new Sum(new Const(2), new Const(3)),
new Const(4)
  );

  StringBuilder sb = new
StringBuilder();
  (expr with InfixExprCollector).
apply(sb);
  String s = sb.toString(); //
"(2 + 3)*4"
```

Таким образом, механизм присоединяемых типажей позволяет модифицировать функциональность экземпляров классов, добавляя к ним методы в зависимости от контекста. В частности, это позволяет реализовать модульное решение проблемы дополнительного метода.

## ОБРАБОТКА СОБЫТИЙ. MVC

В качестве примера использования присоединяемых типажей совместно с обобщенными определениями Java мы рассмотрим реализацию методологии MVC с обработкой событий. Дан-

ное сочетание является характерным для приложений, ориентированных на обработку запросов, поступающих от пользователя, в частности, при разработке графического интерфейса и Web-приложений.

Построение MVC-фреймворка начнем с определения базовых типов модели и вида:

```
class Model<M extends Model<M>> {
      private CompositeView<M>
view;

    public Model(CompositeView<M>
view) {
        this.view = view;
    }

      public CompositeView<M>
getView() {
        return view;
    }
}

class View<M extends Model<M>> {
    private M model;

    public M getModel() {
        return model;
    }

    public void setModel(M model)
{
        this.model = model;
    }
}

class CompositeView<M extends
Model<M>> extends View<M> {
     private Collection<View<M>>
views;

    public CompositeView(Collecti
on<View<M>> views) {
        this.views = views;
    }

      public Collection<View<M>>
getViews() {
        return views;
    }
}
```

Классы `Model` и `View` являются типами моделей и видов соответственно. Класс `CompositeView` представляет собой специальную реализацию вида, инкапсулирующую внутри себя коллекцию других видов. Использование экземпляра `CompositeView` внутри модели позволяет динамически изменять набор ассоциированных с ней видов, добавляя или удаляя их при необходимости.

Мы используем параметр `M` для описания фактического типа модели. Параметризация интерфейсов `Model` и `View` позволяет использовать механизм параметрического полиморфизма Java для достижения ковариантного изменения типа модели в подклассах `Model` и `View`. Это позволяет реализациям вида ссылаться не на базовый тип `Model`, а конкретную реализацию модели, что является важным обстоятельством, так как логика реализации вида, как правило, существенно зависит от соответствующей модели. Заметим, что мы не параметризуем эти классы типом вида, так как реализация модели, вообще говоря, не должна зависеть от связанных с ней видов (особенно учитывая тот факт, что множество видов конкретного экземпляра модели может меняться с течением времени).

Основное назначение вида состоит в реализации обработки событий, генерируемых моделью. Мы используем следующую форму базового класса событий `Event` (как и другие классы фреймворка, он параметризуется типом модели `M`):

```
class Event<M extends Model<M>> {
    private M model;

    Event(M model) {
        this.model = model;
    }

    public final M getModel() {
        return model;
    }
}
```

Далее необходимо связать объекты `View` с обработкой событий. Так как логика обработки события специфична для каждого конкретного события, поэтому обработчик должен получать на вход объект соответствующего подкласса `Event`. По этой причине нецелесообразно включать непосредственно в класс `View` базовый метод обработки события типа `handle(Event e)`, так как тогда подклассы `View` будут вынуждены самостоятельно выполнять диспетчериза-

цию вызова метода handle() для отдельных классов событий. Кроме того, это сделает в дальнейшем невозможным отделение реализации вида от логики обработки конкретных событий. Следовательно, мы должны ввести собственный тип для обработчика события, параметризуя его типом события:

```
trait Handler<M, E extends Event<M>>
for View<M> {
    public void handle(E event);
}
```

В этом случае вид должен хранить коллекцию обработчиков. Проблема состоит в том, что система типов Java не позволяет адекватно описать ее тип:

— Collection<Handler<Event>> не позволяет добавлять в коллекцию реализации Handler, параметризованные подклассами Event — в силу инвариантности параметризованных типов по отношению к подтипам.

— Collection<Handler<? extends Event>> решает эту проблему, но не позволяет вызывать у элементов коллекции метод handle(), так как параметр E находится в нем в контравариантной позиции, в то время как ограничение extends Event подразумевает ковариантность параметризованного типа.

— Collection<Handler<? super Event>> обладает тем же недостатком, что и Collection<Handler<Event>>.

Решение, основанное на присоединяемых типажах, состоит в отказе от явного хранения коллекции обработчиков. Диспетчеризация выполняется автоматически во время выполнения программы — для обработки события достаточно описать типаж, реализующий Handler<MyEvent> в контексте MyView, где MyEvent — класс обрабатываемого события, а MyView — класс вида. Для запуска процедуры обработки события необходимо присоединить типаж Handler<MyEvent> к объекту вида и вызвать метод handle(), передав ему экземпляр события MyEvent.

Поясним сказанное на простом примере MVC-модели текстового редактора, способного загружать текст из файла, модифицировать его, сохранять в файл и позволяющий реагировать на соответствующие события.

В качестве модели будет выступать класс Document, инкапсулирующий текстовую строку (для краткости здесь мы не приводим полный код реализации):

```
class Doc extends Model<Doc> {
    private String content = "";

    public Doc(CompositeView<Doc>
view) {
        super(view);
    }

   public void loadFromFile(String
fileName) {
        // ...
       (with Handler<LoadEvent>).
handle(new LoadEvent(this));
    }

    public String getContent() {
        return content;
    }

    public void setContent(String
content) {
        this.content = content;
       (with Handler<OpenEvent>).
handle(new OpenEvent(this));
    }

    public void saveToFile(String
fileName) {
        // ...
       (with Handler<SaveEvent>).
handle(new SaveEvent(this));
    }
}
```

В качестве вида используем класс Editor:

```
class Editor extends View<Doc> {
   // ...
}
```

Для описания события загрузки документа из файла опишем событие LoadEvent.

```
class LoadEvent extends Event<Doc>
{
    private String fileName;

     LoadEvent(Doc model, String
fileName) {
        super(model);
        this.fileName = fileName;
    }

    public String getFileName() {
        return fileName;
    }
}
```

Аналогичное событие SaveEvent используем для сохранения документа. Кроме того, изменение текста описывается событием ChangeEvent:

```
class ChangeEvent extends Event<Doc>
{
      private String oldContent;

      ChangeEvent(Doc model, String
oldContent) {
          super(model);
               this.oldContent =
oldContent;
      }

      public String getOldContent() {
          return oldContent;
      }
}
```

Ниже представлено описание типажа, выполняющего обработку события OpenEvent (обработчики остальных событий определяются аналогично).

```
public trait EditorOpenHandler
extends Handler<OpenEvent> for
Editor {
     public void handle(OpenEvent
event) {
          // ...
}
}
```

В данном случае обработчики описаны вне класса Editor. Мы также могли бы закрыть код обработчиков внутри класса Editor. Преимущество открытого описания типажей состоит в возможности переопределения связанного с ними аспекта поведения — клиентский код может определить собственный подтип для, скажем, EditorOpenHandler и перекрыть его метод handle(), изменив тем самым реакцию на событие OpenEvent. Никаких изменений в существующих классах при этом не потребуется.

## ЗАКЛЮЧЕНИЕ

В данной статье мы представили один из вариантов расширения языка Java, основанный на механизме присоединяемых типажей. Описанное расширение позволяет использовать на платформе Java гибкую форму наследования, при которой функциональность класса и его связи наследования могут меняться в зависимости от контекста. Рассмотрены необходимые изменения в синтаксисе, системе типов Java, ключевые аспекты семантики присоединяемых типажей, а также основные области их применения.

### СПИСОК ЛИТЕРАТУРЫ

1. Clifton C.et al. MultiJava. Design Rationale, Compile Implementation and Applications — TR#04-01b, Department of Computer Science, Iowa State University, 2004. http://multijava.sourceforge.net/

2. James Gosling. The Java™ Language Specification. Third Edition. / James Gosling, Bill Joy, Guy Steele, Gilad Bracha — Addison Wesley, 2005. — 688 pp.

3. Millstein T.et al. Expressive and Modular Predicate Dispatch for Java // — ACM Transactions on Programming Languages and Systems. — 2009. — Vol. 31, No. 2, Article 7.

4. Odersky M. The Scala Language Specification. Version 2.7 — 2009. http://www.scala-lang.org/docu/files/ScalaReference.pdf

5. Тюкачев Н. А., Седунов А. А. Объектно-ориентированный метаязык // Вестник ВГУ, Системный анализ и информационные технологии. — 2009. — № 1. — С. 122—132.

**Седунов Алексей Александрович** — ассистент каф. "Программирование и информационные технологии" Воронежского государственного университета, тел. 8-961-182-31-28, e-mail: alexey.sedunov@gmail.com.

**Sedunov A. A.** — Post-Graduate Student, the dept. of Programming and Informational Technologies, Voronezh State University. Tel.: (4732) 208-470, e-mail: alexey.sedunov@gmail.com.