DOI: 10.15514/ISPRAS-2019-31(5)-6

Разработка языка: OOP or not OOP or better OOP

А.Е. Недоря, ORCID: 0000-0001-8998-7072 <nedoria.aleksei@huawei.com>
Huawei Technologies Co., Ltd, Russian Research Institute
191119, Россия, Санкт-Петербург, улица Марата, 69-71

Аннотация. В рамках процесса совершенствования экосистемы разработки приложений для различных устройств Huawei компания работает над новым языком программирования. В статье рассматривается подход к реализации ООР в языке программирования, который рассматривается как движение в сторону компонентно-ориентированного программирования.

Ключевые слова: языки программирования; компонентно-ориентированное программирование; объектно-ориентированное программирование; программная экосистема.

Для цитирования: Недоря А.Е. Разработка языка: OOP or not OOP or better OOP. Труды ИСП РАН, том 31, вып. 5, 2019 г., стр. 89-94. DOI: 10.15514/ISPRAS-2019-31(5)-6

Language Design: OOP or not OOP or better OOP

A.E. Nedoria, ORCID: 0000-0001-8998-7072 <nedoria.aleksei@huawei.com>
Huawei Technologies Co., Ltd, Russian Research Institute,
69-71, Marata street, St. Petersburg, 191119, Russia

Abstract. As part of the process of improving the application development ecosystem for various Huawei devices the company is working on a new programming language. The principal feature of the new language is the support of component-oriented programming (COP), by which we understand the possibility of assembling (an essential part) of the program from ready-made components. One of the steps in the direction of COP is, from our point of view, the right choice of OOP features. In the current work, we do not consider COP directly, focusing on the OO paradigm implementation. Currently, the situation with the OO paradigm is quite confusing. In fact, there is no consensus in the IT community on what OOP is. Suffice it to note that OOP in Go and Rust is fundamentally different from OOP in C++ and Java. Languages with object orientation based on classes and implementation inheritance (CLOP languages, where CLOP – Class-Oriented Porgramming) are criticized for the lack of flexibility and for the problems of developing reusable components. As component's reusing is important for us, we propose OOP features that are non-CLOP and allows one to implement objects that can be extended (by adding methods) without the need to make changes to the source code of the object and with minimal recompilation of clients.

Keywords: programming language; component-oriented-programming; object-oriented programming; class-oriented programming; software ecosystem.

For citation: Nedoria A.E. Language Design: OOP or not OOP or better OOP. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 5, 2019, pp. 89-94. (in Russian). DOI: 10.15514/ISPRAS-2019-31(5)-6

1. Введение

В рамках процесса совершенствования экосистемы разработки приложений для различных устройств Huawei компания работает над новым языком программирования.

Принципиальной частью экосистемы должна быть унифицированная, дружелюбная к разработчикам среда разработки, обеспечивающая разработку мульти-платформенных приложений и высокий уровень повторного использования.

Работа над языком и экосистемой идет в нескольких направлениях:

- сбор и анализ требований разработчиков;
- продумывание метрик и разработка системы измерений, позволяющей с удовлетворительной степень объективности сравнивать характеристики экосистемы и приложений, в том продуктивность разработчиков и эффективность приложений в широком смысле, включая производительность, потребление энергии, памяти и т.п.;
- разработка прототипов языка, компилятора, библиотек и среды исполнения.

Принципиальной чертой нового языка мы видим средства компонентно-ориентированного программирования (COP – Component-Oriented Programming) [1], под которым мы понимаем возможность сборки (существенной части) программы из готовых компонент. Одним из шагов в направлении COP является, с нашей точки зрения, правильный выбор средств ООР.

2. Подход к ООР

В текущей работе мы не рассматриваем COP напрямую, сосредоточившись на ОО парадигме. Заметим, что в настоящее время, ситуация с ОО парадигмой весьма запутанная. По сути, в IT сообществе отсутствует общее понимание того, что такое ООР. Достаточно отметить, что ООР в языках Go [2, 3] и Rust [4,5] принципиально отличается от ООР в языках C++ и Java. Если добавить к списку языков Lua [6,7] и EO (Elegant Objects) [8,9], мы увидим, что словом ООР называются существенно разные и часто противоположные подходы. Для иллюстрации приведем сравнительную таблицу конструкций языков (табл. 1):

Табл. 1. Сравнение конструкций объектно-ориентированных языков
Table 1. Comparison of constructions of object-oriented languages

Конструкция	C++	Java	Go	Rust	EO	Lua
Классы	+	+	-	-	-	-
Методы и атрибуты класса	+	+	-	-	-	-
Интерфейсы	-	+	+	+	+	-
Наследование абстракций или интерфейсов	+	+	-	+	+	-
Наследование реализации	+	+	-	-	-	-
Явная поддержка неизменяемости (immutability)	-	-	-	+	+	-
Динамическое создание классов	-	-	-	-	-	+ (аналогов классов)

Заметим, что для C++/Java наследование реализации является принципиальным и необходимым механизмом, а авторы Go, Rust и EO считают этот механизм недопустимым.

Впрочем, не будем рассуждать об ООР вообще, вспомним перечисленные выше требования к языку, которые включают продуктивность разработчика и поддержку повторного использования.

Как известно, языки, в которых объектно-ориентированность основана на классах (для этого варианта ОО будем использовать аббревиатуру CLOP – Class-Oriented Programming [10]), критикуются (в том числе) за отсутствие гибкости и за проблемы разработки повторно используемых компонент [11,12,13,14,15,16]. Приведем известную цитату Джо Армстронга (Joe Armstrong), автора языка Erlang:

> *I think the lack of reusability comes in object-oriented languages, not in functional languages. Because the problem with object-oriented languages is they've got all this implicit*

environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.[1]

Источником проблемы является наследование реализации. Как мы видим, ОО языки нового поколения отказываются от классов и от наследования реализации. И, тем самым, позволяют избавиться или сделать менее острой проблему добавления горилл и джунглей к банану.

Понимая это, мы можем сделать первый шаг к определению ООР в новом языке – это **отказ от CLOP и от наследования реализаций**.

Второй шаг следует из понимания того, что возможность повторного использования компонентов **(compile once, use everywhere)** для разных устройств существенно увеличивает эффективность СОР. Частным следствием этого понимания, является **требование расширения объектов** (добавления методов) без необходимости внесения изменений в исходный код и с минимизацией перекомпиляции.

Рассмотрим пример. Определим структуру данных `List` с методами `Insert` и `Remove` (в некотором условном синтаксисе):

```
List = struct {
   fn (l: list) Append(e: Element) …
   fn (l: list) Remove(e: Element) …
}
```

Предположим, что некоторым приложения, использующим `List`, нужна операция добавления списка к списку. Рассмотрим способы реализации.

Способ 1. Операция реализована функцией, внешней по отношению к объекту `List`: `fn AppendList(to: list, from: list)`, использующей `to.Append` и итератор для обхода элементов `from`.

В такой реализации есть несколько недостатков:

1. функцию `AppendList` надо импортировать дополнительно (и знать о её существовании);
2. отсутствие однородности: вызов функции `AppendList` отличается от вызова `Append`;
3. Производительность отдельной функции `AppendList` скорее всего, хуже, чем метода, в котором можно использовать детали реализации.

Способ 2. Вписать `AppendList` в `List`. Это избавляет от перечисленных выше недостатков, но приводит к изменению исходного кода и к необходимости перекомпиляции всех программных частей, использующих `List`.

Оба этих способа нам не подходят, мы хотим обеспечить расширение, не компрометируя производительность и без необходимости перекомпиляции тех частей, которые не используют `AppendList`.

Способ 3. Предлагаемое решение основано на известном подходе: чтобы объединиться, надо решительно размежеваться. Выделим следующие атомарные (отдельно компилируемые) сущности:

- определение `List` как скрытого типа;
- структура данных `List:impl`, которая определяет способ реализации списка – используя массив или односвязный список или что-то третье;

[1] Перевод: Я думаю, что отсутствие возможности повторного использования свойственно объектно-ориентированным, а не функциональным языкам. Проблемой объектно-ориентированных языков является то, что с ними всегда связана неявная среда. Вы хотели банан, а получаете гориллу с бананом и все джунглями в придачу.

- отдельно каждый метод списка: `Append`, `Remove`. Для реализации каждого метода используется `List:impl`.

Теперь соберем «объект» из составных частей с помощью объединяющей конструкции, которую мы называем «`usebox`»:

```
usebox std.containers.list
export List:impl as List + Append + Remove;
```

После этого разработчик, который импортирует `std.containers.list`, может использовать `List` общепринятым способом: `var l: List; l.Append() …`

Для добавления метода `AppendList` его надо реализовать (как отдельную единицу компиляции, использующую `List:impl`), а после этого сделать новый `usebox`:

```
usebox std.containers.list2
export List:impl as List + Append + Remove + AppendList;
```

Теперь во всех программные частях, которым нужен `AppendList`, надо изменить импорт и перекомпилировать их. Те же части, которые используют только `Append` (и первый `usebox`), не требуют изменений и перекомпиляции. Кроме того, отсутствует дублирование кода, так как `usebox` – это конструкция времени компиляции. В рамках одного приложения могут использоваться `std.containers.list` и `std.containers.list2`, но код методов не будет дублироваться.

Предлагаемый механизм противоречит привычной инкапсуляции, и, на первый взгляд, уменьшает надежность программ за счет возможности доступа к внутреннему устройству `List`.

В действительности, мы просто привыкли и не замечаем странностей привычной (статической) инкапсуляции. Вообще говоря, вместо механической защиты от «всех», защищать внутреннее устройство надо от тех, кто может его испортить. Или от тех, кто не авторизован.

Для описанных атомарных сущностей можно выделить несколько уровней доступа:

- доступ к скрытому (абстрактному типу) `List` – доступно для всех разработчиков;
- доступ к коду метода – для разработчика метода;
- доступ к реализации структуры данных для использования, но не для изменения – для разработчиков методов;
- доступ к реализации структуры данных для изменения – для владельца кода (code owner).

Так как в современной разработке всегда используются системы управления кодом и версиями, технически мы готовы к переходу **к защите кода через авторизацию**.

3. Заключение

Мы описали одну из черт нового языка, полезность которой сейчас проверяется с использованием прототипа компилятора и среды исполнения. Предлагаемое решение во многом опирается на предыдущие работы автора, см. [17,18,19,20,21].

Список литературы / References

[1]. Szyperski C. Component Software: Beyond Object-Oriented Programming. Addison-Wesley Professional, 2002, 411 p.

[2]. The Go Programming Language Specification. Available at: https://golang.org/ref/spec, Version of July 31, 2019, accessed 09.10.2019.

[3]. Lukac L. Is Go an Object Oriented language? Avaliable at: https://medium.com/gophersland/gopher-vs-object-oriented-golang-4fa62b88c701, accessed 09.10.2019.

[4]. Klabnik S., Nichols C. The Rust Programming Language, Available at: https://doc.rust-lang.org/book/title-page.html, accessed 09.10.2019.

[5]. Klabnik S., Nichols C. Object Oriented Programming Features of Rust. Avaliable at: https://doc.rust-lang.org/book/ch17-00-oop.html, accessed 09.10.2019
[6]. Lua 5.3 Reference Manual. Avaliable at: https://www.lua.org/manual/5.3/, accessed 09.10.2019.
[7]. Lua. Object Orientation Tutorial. Avaliable at: http://lua-users.org/wiki/ObjectOrientationTutorial, accessed 09.10.2019.
[8]. EO, The programming language, Available at: https://github.com/yegor256/eo, accessed 09.10.2019.
[9]. Бугаенко Е. Элегантные объекты. Java Edition. Питер, Санкт-Петербург, 2019, 224 стр. / Bugaenko E. Elegant objects. Java Edition. Piter, St. Petersburg, 2019, 224 p.
[10]. Недоря А.Е. CLIP/CLOP vs pure OOP. / Nedoria A.E. CLIP/CLOP vs pure OOP. Avaliable at: http://xn--80aicaaxfgwmwf3q.xn--p1ai/?p=152, accessed 09.10.2019 (in Russian).
[11]. West D. Object Thinking. Microsoft Press, 2004, 368 p.
[12]. Suzdalnitski I. Object-Oriented Programming – The Trillion Dollar Disaster. Avaliable at: https://medium.com/better-programming/object-oriented-programming-the-trillion-dollar-disaster-92a4b666c7c7, accessed 09.10.2019.
[13]. Scalfani C. Goodbye, Object Oriented Programming. Avaliable at: https://medium.com/@cscalfani/goodbye-object-oriented-programming-a59cda4c0e53, accessed 09.10.2019.
[14]. Armstrong J. Why OO Sucks. Avaliable at: http://harmful.cat-v.org/software/OO_programming/why_oo_sucks, accessed 09.10.2019.
[15]. Will B. Object-Oriented Programming is Bad. Avaliable at: https://www.youtube.com/watch?v=QM1iUe6IofM, accessed 09.10.2019.
[16]. Church M. Was object-oriented programming a failure? Avaliable at: https://www.quora.com/Was-object-oriented-programming-a-failure/answer/Michael-O-Church?ch=10&share=cb6efe55&srid=XoXvj, accessed 09.10.2019.
[17]. Недоря А.Е. Триада языков программирования. / Nedoria A.E. The triad of programming languages. Avaliable at: http://xn--80aicaaxfgwmwf3q.xn--p1ai/?p=298, published 20.09.2018, accessed 09.10.2019 (in Russian).
[18]. Недоря А.Е. Технология разработки мультиплатформенных программ на основе явных схем программ. / Nedoria A.E. Technology for developing multi-platform programs based on explicit program schemes Available at: http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm, published 04.05.2018, accessed 09.10.2019 (in Russian).
[19]. Недоря А.Е. Компонентный ассемблер для цифрового пространства. / Nedoria A.E. Component Assembler for Digital Space. Available at: http://digital-economy.ru/stati/komponentnyj-assembler-dlya-tsifrovogo-prostranstva, published 05.12.2018, accessed 09.10.2019 (in Russian).
[20]. Недоря А.Е. Компонентный ассемблер. Часть 2. Дух языка. / Nedoria A.E. Component assembler. Part 2. The spirit of language. Available at: http://digital-economy.ru/stati/komponentnyj-assembler-chast-2-dukh-yazyka, published 18.01.2019, accessed 09.10.2019 (in Russian).
[21]. Недоря А.Е. Ворчалки о программировании. / Nedoria A.E. Gruntings about programming. Available at: http://алексейнедоря.рф, accessed 09.10.2019 (in Russian).

Информация об авторах / Information about authors

Алексей Евгеньевич Недоря, к.ф.-м.н., Huawei Russian Research Institute. Сфера научных интересов: языки программирования, программные экосистемы, технологии программирования, мульти-платформенное программирование, компонентно-ориентированное программирование.

Aleksei Nedoria – PhD of Physical and Mathematical Sciences, Huawei Russian Research Institute. Research interests: programming languages, software ecosystems, programming technologies, multi-platform programming, component-oriented programming.