Третья структурная эволюция. Введение в архитектурное программирование

А. Е. Недоря, г. Санкт-Петербург

За годы развития программной индустрии мы были свидетелями двух структурных эволюций. Первая, структурное программирования принята всеми, вторая, модульное программирование находится в процессе принятия даже самыми неповоротливыми сообществами, например, сообществом C++. На мой взгляд, назрела необходимость третьей структурной эволюции, которую я называю архитектурным программированием. Суть её в том, что архитектура должна стать обязательной и неотъемлемой частью любой долгоживущей и развивающейся программной системы. Статья вводит определение и основные понятия архитектурного программирования.

Введение

Предыдущие статьи [1, 2, 3, 4] описывали разработку языка Тривиль и его реализацию. При разработке языка программирования необходимо понимать, что язык — это не самоцель, а инструмент. Естественно, что у каждого инструмента есть область применения и ограничения. В предыдущих статьях много говорилось о целях языка Тривиль и требованиях к нему. При этом цели рассматривались достаточно узкие.

Попробуем теперь посмотреть шире. Как я уже писал в [5], мы хотим делать программы проще, быстрее, с меньшими усилиями и, что очень важно, предсказуемо. То есть, на этапе планирования, мы хотим уметь достаточно точно определить временные рамки проекта и необходимые ресурсы, в том числе, трудозатраты.

Проблема в том, что разработка программных систем (ПС), в одном аспекте, существенно отличается от разработки более материальных продуктов, таких как здания, мосты, станки или процессоры. Для материальных (hard) продуктов есть явное разделение между этапами проектирования, изготовления и эксплуатации. Для программных систем, такого явного разделения, как правило, нет. Программные системы до-проектируются и до-изготавливаются во время эксплуатации. Если доработка программной системы остановилась, то, обычно, это приводит к завершению эксплуатации, к завершению жизненного цикла программной системы. Запомним эту особенность и посмотрим с другой стороны.

Почему мы делаем программы трудно, дорого и непредсказуемо? Очевидно, что нам что-то мешает. На мой взгляд, одной из самых существенных помех является то, что мы **не умеем работать с архитектурой программных систем**. Здесь под “мы”, я подразумеваю всех разработчиков, в том числе, уважаемых мной системных архитекторов, тем более, что я много лет работал именно системным архитектором, начиная с тех времен, когда у этой профессии еще не было названия.

Поясню, что я имею в виду, в самом общем виде.

Рассмотрим разработку достаточно большой ПС. В идеальном случае она начинается с постановки задач, определения требований и построения архитектуры, например, в виде UML диаграмм.

Далее, подключаются разработчики. Но вот беда, у архитектора нет инструментов, чтобы проверять, что то, что пишут разработчики, соответствует архитектуре, кроме просмотра кода. Это может показаться не очень большой проблемой. Например, при строительстве дома легко увидеть, что труба с водой проведена не в ванную, а в спальню. К сожалению, с программным кодом, это не просто, тем более что на одного системного архитектора приходится далеко не один разработчик. В итоге, даже на первом этапе, код только приблизительно соответствует архитектуре.

Дальше все становится только хуже. Смена архитектора проекта, как правило, приводит, к тому, что новый архитектор, во-первых, не полностью в курсе того, что задумывал предыдущий, а во-вторых, у него свои взгляды на архитектуру. Так же пагубно на соответствие кода архитектуре действует необходимость внесения модификаций.

В процессе развития ПС, архитектура все больше размывается. И увы, чем лучше ПС, чем больше

она используется, тем больше деградация. В коде появляются связи, которые не были предусмотрены архитектурой, дублирующие куски кода, не используемые куски кода, и так далее. Чем дальше, тем дороже и не безопасней становится доработка кода, больше технический долг.

Разные подходы предлагались и предлагаются для лечения этой болезни. На мой взгляд, эта болезнь их тех, которые не надо лечить, а надо создать условия, в которых болезнь не может появится.

Для этого, на мой взгляд нужна **третья структурная эволюция**.

Структурные эволюции

Говоря о первой и второй структурных эволюциях[1], я имею в виду переходы

1. к структурному программированию
2. к модульному программированию

Началом структурного программирования считается выход статьи Эдсгера Дейкстры “Go to statement considered harmful” в журнале Communications of the ACM за 1968 год. За прошедшие 50 с лишним лет, структурное программирования стало де-факто стандартом для языков программирования.

Для модульного программирования нет такой явно выделенной точки начала, но этот эволюционный шаг очевидно связан с Никлаусом Виртом, его языками Модула[6] и Модула-2 и их предшественником, языком Mesa. Достаточно условно, мы можем считать середину 1970-х началом второй структурной эволюции. В отличии от первой эволюции, вторая эволюция до сих пор не стала общепринятой, хотя это потихоньку происходит. Во многих языках поддержка модульного программирования появилась с самого начала, например, в языках Python, OCaml, Rust и Go, а в другие добавлялась по ходу развития языка, например, в языки Objective-C (2013.) или C++20 (2020).

Первая структурная эволюция определила структуру на уровне операторов, вторая на уровне достаточно больших частей кода, называемых модулями или пакетами, **третья структурная эволюция**, переход к **архитектурному программированию**, должна ввести структуру на уровне программной системы.

Полагаю, что многие читатели возразят, что система модулей через импорт задает структуру всей программы. Далее в статье мы рассмотрим, почему нам нужна другая структуризация.

Архитектурное программирование

Суть третьей структурной эволюции в том, что **архитектура должна стать обязательной и неотъемлемой частью программной системы**.

Наверно, этого можно достичь более чем одним способом, но я вижу и рассматриваю способ, основанный на следующих принципах:

- **Архитектура** программной системы описана **явно** на некотором языке
- Описание архитектуры является обязательными входными данными для инструментов (компиляторы, системы сборки), строящих исполняемый код программной системы.
- Архитектура задает устройство системы, составные части и взаимодействие между ними.

Еще раз уточню, в архитектурном программировании (АРП) программную систему **нельзя** построить без описания архитектуры. Описание архитектуры отображается (прямо или косвенно) в код программной системы. Любая модификация (кроме локальной модификации) требует изменения архитектурного описания.

Архитектурное программирования является новой технологией, своего рода, Terra Incognita. Для того чтобы работать на этой "Неизвестной земле", мы должны тщательно продумать терминологию. Заимствование существующей, пропитанной привычной семантикой терминологии будет мешать видеть новое и думать. Я считаю неверным использовать, например, термины *компонента* или *модуль* для составных частей программной системы. Эти термины определяются в разных языках и системах по-разному, однозначных определений не существует.

Поэтому, в дальнейшем тексте я буду использовать новые термины:

- **Архитектурная схема**, для обозначения исходного описания архитектуры.

---

1 Я намеренно использую слово “эволюция”. Революций не было, было медленное принятие структуризации.

- **Арка** (архитектурная компонента), для обозначения составной части ПС.

Переформулирую то, что уже было сказано: **архитектурная схема** определяет
- **устройство** программной системы, как правило, это дерево арок или дерево деревьев арок;
- **и взаимодействие** между арками. Взаимодействие арок полностью определяется схемой, арки не могут взаимодействовать напрямую, помимо схемы.

Следствие принципов АРП является то, что ПС программируется минимум на 2-х языках: на языке архитектурной схемы и языке, на котором пишутся арки. Впрочем, я полагаю, что арки могут и будут писаться на разных языках.

Прежде чем перейти к более детальному описанию, приведу пример, чтобы на нем показать отличия АРП от привычных технологий и дать материал для дальнейшего обсуждения.

Архитектурный “Привет, мир!”

Рассмотрим классический “Пример, мир!”, сделанный в технологии АРП. Замечу, что для такой программы АРП является избыточным, целевая область АРП — это большие, развивающиеся, долгоживущие программные системы.

В примере используются два языка программирования: Арс — язык программирования **Ар**хитектурных **с**хем и Арвиль (Архитектурный Тривиль):
- Язык **Арс** — это (почти) декларативный язык. Первым прототипом его был язык описания схем в среде Вир [7], работа над вторым прототипом велась в конце 2023 года и первой половине 2024 года. К середине 2024 года мне стало понятно, во многом благодаря общению с коллегами[2], что язык надо существенно пересматривать, что сейчас и происходит. Примеры на Арсе являются предварительными, язык сейчас активно прорабатывается и меняется.
- Язык **Арвиль** является строгим надмножеством языка Тривиль, к которому добавлены конструкции, необходимые для разработки арок. Он достаточно стабилен, но может измениться под влиянием изменений в языке Арс.

	Архитектурная схема: **Арс**		Описание арки: **Арвиль**
1	`арс {`	1	`арка пример`
2	`  мир: арка(адрес: "a2::примеры/пример") {`	2	
3	`  протокол { фн привет(имя: Строка) }`	3	`импорт "стд::вывод"`
4	`  вход привет("мир")`	4	
5	`  }`	5	`фн (_) привет*(имя: Строка) {`
6	`}`	6	`  вывод.ф("Привет, $;!\n", имя)`
		7	`}`

Для создания программы, сейчас, на этапе прототипирования, используется двухэтапная схема:
- сначала запускается Арс компилятор, который строит код на Тривиле,
- потом Тривиль компилятор делает исполняемую программу из исходника, полученного из схемы и исходника арок:

```
D:\0-Projects\trivil-0>арс a2/схемы/2
арс (v0.1): компилятор языка Арс (v0.1)
сохранено a2/схемы/2/арп.tri
Без ошибок

D:\0-Projects\trivil-0>трик a2/схемы/2
трик (v0.78): компилятор языков Тривиль (v0.9.4) и Арвиль (v0.2)
Выполнить: ./арп  Собрать: ./_си/build.bat
Без ошибок

D:\0-Projects\trivil-0>арп
Привет, мир!
```

Программа печатает, как и ожидается: `Привет, мир!`

2 Особые благодарности коллегам А. Канатову, Е. Зуеву и Д. Соломенникову.

Разберем пример подробнее. Текст на Арс задает устройство программы. В примере, программа состоит из двух вложенных **экземпляров** арок. Арка (архитектурная компонента) — это тип, некоторый (не точный) аналог класса.

Первый экземпляр арки записан в строках 1-6, второй, вложенный, в строках 2-5. Рассмотрим определение второго экземпляра, который называется “мир”:

- это экземпляр арки, у которой задан статический адрес описания "a2::примеры/пример";
- для нее задан протокол, который должна реализовать арка (это требование или constraint);
- и начальное действие (вход): вызов метода `привет("мир").`

Для первого экземпляра ничего не задано, для него будет использована стандартная арка-контейнер, которую, с легкой руки А. Канатова, я называю **аркада**[3].

Выполнение программы на Арс состоит из нескольких этапов:

- создание и инициализация экземпляров основной схемы[4],
- подключение связей (в примере связей нет),
- запуск: выполнение всех входов.

В нашем примере, исходники всех используемых арок определены статически. Компилятор может проверить требования (constraints) и сгенерировать оптимальный код. Для АРП статическое определение арки является одним из вариантов, другие варианты:

- подключение кода арки и проверка требований во время исполнения[5],
- или во время сборки, при этом проверка требований может быть как во время сборки (AOT), так и во время исполнения (JIT).

Все эти варианты нормальные или штатные для АРП. Архитектурная схема задает “шаблон” программы. Например, достаточно изменить адрес экземпляра вложенной арки, и вместо вывода текст на экран, может сработать запуск стаи беспилотников, которые огнями выложат в небе указанный текст.

Посмотрим теперь на описание арки:

	Описание арки: **Арвиль**
1	**арка** пример
2	
3	**импорт** "стд::вывод"
4	
5	**фн** (_) привет*(имя: Строка) {
6	вывод.ф("Привет, $;!\n", имя)
7	}

Это модуль (единица компиляции) особого вида. Как уже упоминалось, **арка** похожа на класс, для которого, в этой арке, определен единственный метод `привет`. Имя получателя в методе не используется, и, поэтому, задано символом ‘_’ в строке 5 (используется общепринятое соглашение, что символом подчеркивания обозначаются игнорируемые имена). Библиотека вывода в арке импортируется обычным образом, и об этом мы поговорим отдельно.

Добавим в схему еще один экземпляр арки:

	Архитектурная схема: **Арс**

3 Аркада (в строительстве) - это композиция, состоящая из арок.

4 Любая достаточно большая программа собирается, как иерархия схем, экземпляры подсхем будут создаваться по необходимости.

5 В Вире, например, таким образом реализована защита программы. Зарегистрированная программа скачивает при запуске несколько компонент с сервера, а не зарегистрированная работает в “демо” режиме.

```
1  арс {
2    мир:  арка(адрес:  "a2::примеры/пример") {
3      протокол { фн привет(имя: Строка)  }
4      вход привет("мир")
5    }
6    Вася:  арка(адрес:  "a2::примеры/пример") {
7      протокол { фн привет(имя: Строка)  }
8      вход привет("Вася")
9    }
10 }
```

Теперь программа напечатает:

```
Привет, мир!
Привет, Вася!
```

В этой программе используется два экземпляра одной арки. Изменение или перекомпиляция арки не требуется, так как код её не изменился. Замечу, что в общем случае, у разных экземпляров могут быть разные протоколы.

Зачем нужен язык Арс?

Очевидно, чтобы описывать архитектуру программных систем. Но нужен ли для этого отдельный язык? Первое, что напрашивается, это использование чего-то вроде XML или JSON. Тем более, что в Вире использовался XML-подобный язык.

Есть несколько принципиальных соображений, которые привели меня к необходимости делать новый **язык программирования**, а не использовать язык разметки или язык описания данных:

1. Языки типа XML и JSON неудобны и непривычны для программирования, особенно для чтения и понимания. В некоторых случаях, их использование может сделать более удобным визуальный редактор, но есть вещи, которые сложно выразить в визуальном редакторе, но легко записать в тексте.
2. На моем предыдущем опыте, я понимаю, что чисто декларативный язык для схем не достаточен. Во многих случаях небольшой фрагмент императивного кода позволяет гораздо проще и удобнее решить задачу, например, задачу согласования имен или сигнатур. Подробнее об этом будет написано в следующих статьях. Пока же, в качестве аналогии, вспомните о том, как пишутся обработчики событий в языках декларативного UI.
3. Третье соображение, это отсутствие зависимостей. В статье [1] я достаточно подробно писал об этом применительно к Тривилю. Разработка языка с нуля позволяет отработать терминологию, а здесь она особенно важна.

Замечу, что в разработке языка Арса мне практически не на что опираться. Так что приходится по ходу искать синтаксис и выстраивать семантику. Я не удивлюсь, если Арс в окончательном варианте будет совсем не похож на то, что есть сейчас. Поиск продолжается.

Перепишу предыдущий пример, чтобы показать движение в сторону более привычного языка программирования:

Архитектурная схема: **Арс**

```
1  деталь Привет(кому: Строка) =
2    арка(адрес:  "a2::примеры/пример") {
3      протокол { фн привет(имя: Строка)  }
4      вход привет(кому)
5    }
6
7  арс {
8    мир: Привет("мир")
9    Вася:  Привет("Вася")
10 }
```

В данном случае, сначала определяется **деталь**, а схема собирается из заранее описанных деталей.

Зачем нужен Арвиль?

Я полагаю, что арки могут быть написаны на разных языках программирования, кроме того, должны быть составные арки, состоящие из других арок и схемы соединения.

Арвиль нужен для отработки терминологии и базового инструментария. При использовании других языков часть конструкций придется эмулировать. Арвиль должен дать четкое и однозначное понимание того, что нужно и как оно называется.

Арвиль является надмножеством языка Тривиль, то есть любой корректный текст на Тривиле является корректным текстом на Арвиле. Несколько конструкций, нужных в Арвиле было добавлено непосредственно в Тривиль, а именно, функциональные типы и утиные протоколы [8].

Из того, что нет в Тривиле, в Арвиле есть единица компиляции `арка` и конструкции для описания взаимодействия между арок. По ходу развития языка Арс, Арвиль тоже будет меняться.

Взаимодействие между арками

Определение взаимодействия между арками — это ключевой механизм архитектурного программирования. Он должен обеспечить гибкость и гарантированную надежность.

Я приведу только один пример, так как синтаксис и семантика конструкций, определяющих взаимодействие сейчас пересматривается. Полному описанию взаимодействия будет посвящена следующая статья.

Рассмотрим описания арки, которой для работы нужны данные извне:

Описание арки: **Арвиль**

```
1  арка пример3
2
3  импорт "стд::вывод"
4
5  контакт {
6     Имя: запрос (): мб Строка
7  }
8
9  фн (а) привет*(имя: Строка) {
10    вывод.ф("Привет, $;!\n",  а.Имя())
11 }
```

В данном примере, чтобы экземпляр арки работал, нужно подключение к **контакту**, которые называется “Имя”. Этот контакт используется в функции `привет`: `а.Имя()`.

Контакт определен, как **запрос** (функция) с заданной сигнатурой (без параметров, возвращает опциональную строку). Слово **контакт** намекает на аналогию с электронной компонентой. Чтобы компонента работала, на плате должно быть задано подключение к контакту.

Такое подключение задается архитектурной схемой:

```
1  арс {
2     мир:  арка(адрес:  "а2::примеры/пример3") {
3        контакт Имя = "коллеги"
4        протокол { фн привет()  }
5     }
6     вход мир.привет()
7  }
```

В данной схеме контакту “Имя” сопоставляется литеральная строка “коллеги”. То есть при каждом запросе из арки, будет выдаваться один и тот же ответ. Такое константное подключение

является простейшим случаем.

В более сложном случае, контакт может установить соединение между двумя арками:

```
1  арс {
2    имена:  арка(адрес:  "a2::примеры/имена") {
3      протокол { фн запросить_имя(): Строка  }
4    }
5    мир:  арка(адрес:  "a2::примеры/пример3") {
6      контакт Имя = имена.запросить_имя
7      протокол { фн привет()  }
8    }
9    вход мир.привет()
10 }
```

В этом примере, при обращении к контакту из экземпляра арки `мир` будет вызван метод из экземпляра другой арки, которая в схеме названа `имена`. Арс компилятор проверит сигнатуры методов и контактов и, тем самым, корректность соединения.

Обратите внимание, что имя, в отличие от исходного примера, не передается параметром функции `привет`, и арка, к которой подключен контакт может использовать его произвольным образом. Например, обращаясь к контакту несколько раз, и, получая, возможно, разные результаты.

Более подробно взаимодействие, включая, например, понятия **контакт**, **запрос** и **уведомление**, будет описано в следующей статье.

Архитектурное vs. Модульное

Теперь, когда у нас есть начальный фактический материал, мы можем более предметно поговорить о том, почему модули и импорт не могут быть основой третьей структурной эволюции:

1. Импорт модуля, по сути, является подключением поддерева модулей (транзитивного замыкания по импорту). Другими словами, импортируя модуль, мы подключаем к программе неопределенное количество других модулей. И это весьма напоминает известную проблему глубокого наследования в ООП, которую Joe Armstrong описал так: You wanted a banana but what you got was a gorilla holding the banana and the entire jungle[6].
2. У импортированного модуля может быть состояние и оно может меняться другим модулем импортируемым данных. В рамках программы у нас нет гарантий сохранения состояния модуля, если эта гарантия не поддерживается явным кодом.
3. Замена модуля на другой модуль, даже с таким же интерфейсом, не является простой операцией, она может полностью изменить состав модулей программной системы и ее поведение, см. пункты 1 и 2.
4. Архитектура программной системы задана через импорт неявно, и может меняться даже от того, что используется более новая версия модуля.
5. Связи по импорту обрабатываются в статических языках во время компиляции, а во время исполнения используется другой механизм, например, подключение динамической библиотеки (DLL).

Сравним с архитектурным программированием:

1. Добавление экземпляра арки в схему добавляет только этот экземпляр и ничего больше. См. ниже замечание про импорт в арке.
2. Использование экземпляра позволяет установить нужный уровень доступа, например, исключительный или совместный режим использования экземпляра.
3. Замена экземпляра арки на экземпляр арки другого типа (из другого источника) должна оказывать существенно меньше влияния (или вообще не оказывать влияние) на глобальное состояние программной системы. Обеспечение гарантий локальности изменений — это отдельная тема, которая будет рассмотрена в следующих статьях.
4. Архитектурная схема задана явно и не меняется при изменении кода арки, в том числе при использовании новой версии арки.

---

6 См. https://www.johndcook.com/blog/2011/07/19/you-wanted-banana/

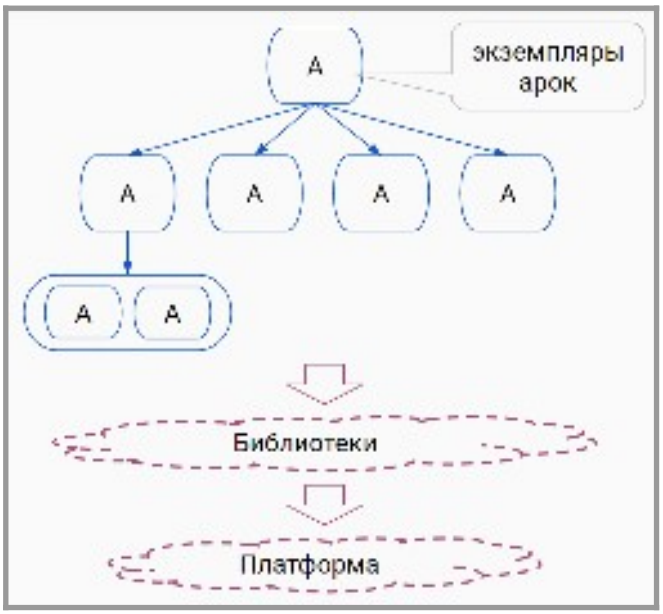
5. Подключение арок и задание связей (взаимодействие арок) могут быть заданы как во время компиляции, так и позже. При этом, задание источника арки и способа взаимодействия во время компиляции рассматривается как оптимизация общего механизма, а не как единственный возможный способ.

В каком-то смысле, можно сказать, что основное отличие арки от модуля в том, что **арка** - это чистый (pure) модуль, по аналогии с чистыми функциями в функциональных языках. Связи (контакты) арки описаны явно, а при подключении каждого экземпляра они явно задаются архитектурной схемой.

Импорт в арке и устройство программы

В примерах на Арвиле используется импорт модулей, кажется, что это противоречит архитектурному подходу. Использование произвольного импорта действительно противоречит импорту и его надо или запретить или наложить существенные ограничения.

В то же время, есть базовая функциональность которая должна быть доступна для арки простым и удобным способом. Очевидный пример: управление памятью. Кроме этого, аркам могут быть нужны операции ввода-вывода, строковые операции, для UI арок нужна поверхность для рисования и работа со шрифтами, и т.д. Часть этой функциональности может обеспечиваться некими системными или платформенными арками, но, на мой взгляд, это не всегда удобно. Скорее я вижу такое устройство программы во время исполнения:



Любая арка имеет доступ к **библиотекам** и **платформе**. Доступ к ним выглядит синтаксически как импорт, хотя может быть реализован через другой механизм[7]. Как платформа, так и библиотеки не должны быть монолитом, и должны подключаться по необходимости, то есть у них должна быть модульная структура.

Ограничения на модули библиотеки и платформы могут быть примерно такие:

- отсутствие состояния (stateless) модуля или корректное обеспечение совместного доступа,
- запрет использования арок из модулей этих слоев, так чтобы через эти модули нельзя было добавить неявное взаимодействие между арками.

Пока это предварительные соображения, которые будут уточнены позже, вместе с механизмом проверка (гарантии) ограничений.

Что дает архитектурное программирование?

Мы можем сделать некоторые промежуточные выводы о тех преимуществах, которые дает архитектурное программирование.

---

7 Например, с помощью динамического подключения, чтобы обеспечить переносимость бинарных файлов между платформами с одним процессором, но разными ОС.

1-й уровень (основное преимущество):

- Архитектура является обязательной и неотъемлемой частью программной системы в течении всей её жизни программной системы.

2-й уровень:

- Новый уровень абстракции — программа, в виде архитектурной схемы, отделена от кода.
- Архитектор делает программу на уровне требований, частей и их взаимодействия, полностью контролируя архитектуру.
- Кардинальное снижение сложности - архитектор работает не с миллионом строк кода, а с описанием, которое в тысячи раз меньше.
- Программист работает не с программой целиком, а с арками, каждая из которых в тысячи раз меньше программы.

3-й уровень:

- Возможность удаления и замены частей программы.
- Возможность добавления точек роста (то есть потенциально развиваемых частей системы).
- Увеличение переиспользования и, следовательно, увеличение унификации программных систем.
- Снижение затрат на модификацию и поддержку программных систем.
- Снижение затрат на подключение к работе над программной системе архитекторов и разработчиков.

Я оставлю этот список на уровне тезисов, давая возможность додумать и критиковать.

Архитектурное программирование и Семейство языков

Архитектурное программирование оформилось в виде идеи во время работы над семейством языков “Языки выходного дня”, и как часто бывает, оказало, в свою очередь, существенное влияние на состав и развитие семейства языков.

Как я сейчас вижу, в семейство должны входит языки следующих категорий:

- Языки разработки архитектурных схем. Первый такой язык, Apc, сейчас разрабатывается.
- Языки разработки архитектурных компонент, первый такой язык — это Арвиль. На мой взгляд, таких языков должно быть несколько. Например, языки, отличающихся по уровню: от языка системного уровня (условно, Rust) до динамического языка (условно, Typescript) и языка для разработки UI. Или языки, отличающиеся набором парадигм или синтаксисом.
- Не архитектурные языки. Пример, Тривиль. Вполне возможно, что нужны будут языки для специальных применений, где архитектурное программирование не работает, например, для разработки драйверов, управления памятью или других специальных, или низкоуровневых задач.

Что дальше?

Задача текущего этапа — доработка языков Apc и Арвиль до уровня прототипа[8], позволяющего разрабатывать программные системы. На уровне прототипа меня вполне устраивает то, что Apc/Арвиль компиляторы выдают код на C99, что приводит к получению монолитной исполняемой программы.

Дальше открывается широкое поле деятельности, включая разработку языка для создания UI, некого правильного аналога LLVM IR, правильной генерации кода, среды исполнения, платформы и так далее.

Впрочем, сейчас об этом говорить несколько рано и надо сосредоточиться на языках Apc и Арвиль, и, в первую очередь, на определении взаимодействия арок. Этому будет посвящена следующая статья.

Литература

[1] Недоря А. Е. Разработка языка Тривиль. Первые шаги к семейству языков. Часть 1, 2024. http://digital-economy.ru/stati/razrabotka-yazyka-trivil-pervye-shagi-k-semeystvu-yazykov-chast-1
[2] Недоря А. Е. Разработка языка Тривиль. Часть 2, 2024. http://digital-economy.ru/stati/razrabotka-

8 Во время написания статьи (февраль 2025) компилятор Apc компилирует только простые примеры.

языка-тривиль-часть-2
[3] Недоря А. Е. Разработка языка Тривиль. Часть 3. Баланс, 2024.
http://digital-economy.ru/stati/разработка-языка-тривиль-часть-3-баланс
[4] Недоря А. Е. Разработка языка Тривиль. Часть 4. Реализация, 2025.
http://digital-economy.ru/stati/разработка-языка-тривиль-часть-4-реализация
[5] Недоря А. Е. Интенсивное программирование, 2022. http://digital-economy.ru/stati/интенсивное-программирование
[6] Wirth, Niklaus. "Modula: a language for modular multiprogramming". *ETH Library*. ETH Zurich. 1976. doi:10.3929/ethz-a-000199440.
[7] Недоря А. Е. Технология разработки мультиплатформенных программ на основе явных схем программ, 2018. http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm
[8] Язык программирования Тривиль. 2024.
https://gitflic.ru/project/alekseinedoria/trivil-0/blob?file=doc%2Fтривиль%2Fописание%2Freport.pdf

References in Cyrillics

[1] Nedorya A. E. Razrabotka yazy`ka Trivil`. Pervy`e shagi k semejstvu yazy`kov. Chast` 1, 2024.
http://digital-economy.ru/stati/разработка-языка-тривиль-первые-шаги-к-семейству-языков-часть-1
[2] Nedorya A. E. Razrabotka yazy`ka Trivil`. Chast` 2, 2024. http://digital-economy.ru/stati/разработка-языка-тривиль-часть-2
[3] Nedorya A. E. Razrabotka yazyka Trivil'. Chast' 3. Balans, 2024.
http://digital-economy.ru/stati/разработка-языка-тривиль-часть-3-баланс
[4] Nedorya A. E. Razrabotka yazyka Trivil'. Chast' 4. Realizaciya, 2025
[5] Nedorya A. E. Intensivnoe programmirovanie, 2022. http://digital-economy.ru/stati/интенсивное-программирование
[7] Nedorya A. E. Tekhnologiya razrabotki mul'tiplatformennyh programm na osnove yavnyh skhem programm, 2018. http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm
[8] Yazyk programmirovaniya Trivil'. 2024.
https://gitflic.ru/project/alekseinedoria/trivil-0/blob?file=doc%2Fтривиль%2Fописание%2Freport.pdf

Недоря Алексей Евгеньевич, к.ф.-м.н.
ORCID 0000-0001-8998-7072
aleksei.nedoria@yandex.ru
Телеграмм канал: t.me/vorchalki_o_prog

Ключевые слова

архитектура программного обеспечения, архитектурное программирование, архитектурная схема, структурная эволюция, семейство языков программирования, разработка языков программирования, модульное программирование

Aleksei Nedoria, The Third Structural Evolution. Introduction to Architectural Programming

Keywords

software architecture, architectural programming, architectural scheme, structural evolution, family of programming languages, programming language development, modular programming

Abstract

Over the years of the software industry development, we have witnessed two structural evolutions. The first, structured programming, is accepted by everyone, and the second, modular programming, is in the process of being adopted by even the most clumsy communities, such as the C++ community. In my opinion,

now we face the need for a third structural evolution, which I call architectural programming. Its essence is that architecture should become a mandatory and integral part of any long-lived and developing software system. The article introduces the definition and basic concepts of architectural programming.