2.3. ИНТЕНСИВНОЕ ПРОГРАММИРОВАНИЕ

Недоря А. Е., к.ф.-м.н., г. Санкт-Петербург

В статья рассматривается состояние программирование через понятия «экстенсивный» и «интенсивный». На основе статистических данных, показан «экстенсивный» характер современного программирования, описан путь перехода к интенсивному программированию и его преимущества.

Если посмотреть темы докладов на ИТ конференциях, то можно прийти к выводу, что программирование, как сфера деятельности, в целом состоялась, осталось немного улучшить. Общение с коллегами показывает, что большинство думает так же или вообще не думает об этом, но есть и исключения. И эти исключения, в том числе и я, полагают, что программирование находится в начале пути развития.

Попробуем посмотреть на программирование, под которым я здесь понимаю индустрию создания программ, через понятия «экстенсивный» и «интенсивный». Термины «экстенсивный» и «интенсивный» обычно относятся к экономическому росту.

«Экстенсивный рост – это увеличение производственных возможностей за счёт расширения количества использованных ресурсов. Интенсивный рост– это увеличение производственных возможностей за счёт более эффективного использования того же количества ресурсов.»

Простыми словами, экстенсивный путь — развитие через увеличение ресурсов, интенсивный — через увеличение производительности при тех же ресурсах.

Развивается ли сейчас индустрия создания программ по экстенсивному или интенсивному пути развития?

Чтобы ответить на этот вопрос, надо определить, что производит программирование. Мое определение: результатом программирования являются **программы**, которые являются **инструментами для решения человеческих задач**. Текстовый редактор — инструмент для работы с документами, социальные сети — инструмент для общения, САПР для проектирования, АСУ для управления и повышения производительности и т.д.

Если так, критерием развития программирования должно быть улучшение инструментов. И дальше можно прикинуть, развитие идет за счет использования большего количества ресурсов или за счет более эффективного их использования?

Я не уверен, что мы можем оценить интегральное качество всех программ или найти подходящие статистические данные. Попробуем использовать косвенные критерии.

Так в исследовании «How much computer code has been written?»[1], написанной в 2020 году, оценивается число строк программного текста, написанного всеми программистами мира за год – дается оценка в 97 миллиардов (97,000,000,000).

Этот объем программного текста является результатом работы 26 миллионов программистов. Столько программистов было по оценке IDC[2] в 2022 году:

"The overall developer population in 2020 was 26.2 million and features 13.5 million full-time developers, 7.8 million part-time developers, and 4.9 million non-compensated developers. Because this data represents the number of developers as of January 1, 2020, the impact of COVID-19 on the worldwide developer population is not reflected in this quantification of the worldwide developer population."

При этом в 2014 году по статистике IDC[3] было 11 миллионов professional software developers, и еще 7.5 миллионов hobbyist software developers, итого 18.5 миллионов.

Простая арифметика показывает, что за 6 лет количество программистов увеличилось на 7.7 миллионов, что дает прирост 1.28 миллионов в год.

Мы видим, что количество используемого ресурса (разработчиков) существенно растет. Растет ли при этом качество выдаваемой продукции опережающими темпами? Сильно в этом сомневаюсь.

Моя субъективная оценка: программирование в последние десятилетия развивается медленно. И это после бурного развития в первые 40-50 лет с появления этого вида деятельности. Мы используем почти те же технологии программирования, что и 30 лет назад. Увеличение эффективности за счет развития технологии практически отсутствует, речь может идти только о единицах процентов.

Замечу еще, что увеличение объема софта вовсе не означает, что добавляется новой (оригинальный, полезный) софт. Я не знаю, есть ли такая статистика, но я уверен, что большая часть якобы нового софта — это клоны уже существующего. Это может быть прямое копирование или переписыва-

1 Sage McEnery. How much computer code has been written? https://medium.com/modern-stack/how-much-computer-code-has-been-written-c8c03100f459

2 IDC Market Perspective quantifies and segments the worldwide developer population in 2020. https://www.idc.com/getdoc.jsp?containerId=US47513521

3 IDC Study: How Many Software Developers Are Out There? https://www.infoq.com/news/2014/01/IDC-software-developers/

ние существующего софта на другой язык программирования или адаптация его к новым условиям (новая ОС, новая VM, новое оборудование, и т.д.).

Ежегодная огромная прибавка объема программного текста и увеличение количества разработчиков, не приводит к существенному увеличению качества существующих программных продуктов или к появлению существенно новых продуктов.

Увы, вывод неутешительный — почти весь «пар уходит в свисток». Думаю, что этот факт подтвердит любой профессионал, знающий, как трудно найти то, что можно использовать в своем проекте. В этой огромной горе произведенного софта, конечно же, есть жемчужины, но как их найти? И как приспособить к своим задачам?

На мой взгляд, приведенные факты говорят о том, что сегодня программирование развивается по экстенсивному пути: быстро растет количество разработчиков, пишется огромное количество программного текста, существенная часть которого, по сути, дублирует ранее существующий.

Давайте подумаем, а как могло бы быть в мире победившего интенсивного программирования…

Полагаю, что в том чудесном мире задача разработки программного кода начинается с поиска готового программного продукта. Если подходящий продукт есть, задача решена. Если нет, надо его построить. А для того, чтобы построить быстро и надежно, желательно максимально использовать готовые части (далее буду использовать более привычное слово «компонента»), которые можно взять из неких хранилищ, и добавить к программному продукту новые компоненты, сделанные специально для него.

Пока, кажется, мало чем отличается от текущего подхода. Впрочем, сразу добавлю: все новые компоненты, которые прошли проверку на возможность переиспользования, должны быть добавлены в хранилище компонент.

Я буду рассматривать только технологические аспекты интенсивного программирования, пропуская вопросы типа: кто кому платит, лицензии и т. д.

Рассмотрим требования, которым должны удовлетворять те компоненты, которые могут быть многократно использованы:

1. независимость от исполняющей платформы;
2. независимость от средств разработки, в том числе от языка программирования;
3. наличие спецификации и возможности формальной проверки соответствия компоненты спецификации;
4. и возможность гибкой настройки или подстройки.

По сути, первые два требования — это требование максимальной прикладной универсальности, то есть, если код написан, то он работает везде, где может. Но перед тем, как рассмотреть подробнее эти требования, подумаем — а что хранится в Хранилищах?

Очевидно, не бинарный код, см. первое требование, и не исходный текст, см. второе требование. На мой взгляд, хранится должен промежуточный код. То есть нечто уже семантически корректное и упрощенное, но не привязанное ни к исходному языку программирования, ни к бинарному коду какой-то платформы.

Это может быть нечто вроде LLVM IR[4] или Rust MIR[5]. Скорее второе, так как одним из присущих LLVM недостатков является то, что IR код сильно завязан на специфику конкретной платформы, для которой он построен (см. TargetMachine).

Промежуточный код позволяет достичь того, что заявляли разработчики Java в 1995 году – “write once, run anywhere”, но без ограничений, накладываемых виртуальными машинами (VM). Промежуточный код перед выполнением транслируется или в код целевого компьютера или в код целевой VM.

Независимость от исполняющей платформы

Другими словами, у компоненты (в промежуточном коде) не должно быть жестких привязок к платформе, а именно:

- к аппаратной части платформы — к процессору, точнее к исполнителю
- и к программной платформе — к ОС и библиотечному окружению.

А это означает, что библиотечное окружение должно быть унифицировано, точнее должен быть унифицирован его API (SysAPI). И еще, не должно быть статической связи (статической линковки) с SysAPI.

Почему пишу про «жесткую» привязку? Потому что некоторая привязка должна быть, например, программу управления телескопом нет смысла запускать без телескопа. Или переформулируя для компоненты, без наличия API телескопа.

Таким образом для каждой компоненты может быть определено (в её спецификации) набор требований, из которых собирается набор требований программного продукта.

В этот набор требований могут входить требования к исполнителю, например, плавающая арифметика – 128 бит, или пропускная способность сети не меньше, чем X. Или требования к SysAPI и тре-

4 LLVM Language Reference Manual. https://releases.llvm.org/8.0.1/docs/LangRef.html
5 Niko Matsakis, “Introducing MIR”, https://blog.rust-lang.org/2016/04/19/MIR.html

бования к наличию других (обслуживающих) компонент — то есть тех компонент (точнее интерфейсов), без которых данная не может работать.

Независимость от средств разработки

Смысл очевидный, компонента может быть использована в составе программного продукта, написанного на другом или на разных языках. Нужно обеспечить возможность сосуществования компонент, написанных на разных языках. Это, как минимум, унификация ABI (application binary interface). Но этого мало, компоненты должны использовать общие программные сервисы, например, сборщик мусора, обработку ошибок, логи и тому подобное. Иначе, в программной системе, состоящей из тысяч компонент, будет тысяча экземпляров сборщика мусора, и это убьет любой вычислитель. Замечу, что тысяча компонент, это нормальная ситуация для программы, сделанной в среде разработки Вир[6].

Эта мысль приводит нас к пониманию необходимости унификации среды исполнения (run-time system). И далее, возникает естественный вопрос — можно ли впрячь в одну телегу коня и трепетную лань? А точнее табун коней, оленей, ежей и ужей…

На мой взгляд нельзя, попытка сделать унифицированный run-time даже для малого набора современных языков программирования приводит к неподъемным монстрам, слишком эти языки разные. Единственный способ, на мой взгляд, это разработка нового семейства языков, которые изначально построены с требованием совместимости.

Почему нужно семейство языков, а не один язык? Потому что в рамках одного языка невозможно удовлетворить разные требования, например продуктивность разработчика и, одновременно, надежность и производительность. А еще у разработчиков могут быть «стилистические» предпочтения к синтаксису языка, и это тоже важно — пусть расцветает сто цветов.

На мой взгляд, семейство должно включать языки, в таких категориях:

- язык, с ручным управлением памятью (типа Rust);
- язык с управлением памятью на основе ARC (automatic reference counter);
- язык со сборкой мусора;
- «скриптовый язык» с динамической типизацией и сборкой мусора.

Подробнее о семействе языков[7].

Как я уже упоминал, языки должны быть совместимыми в смысле возможности исполнения в рамках гибридных программных систем. А для этого нужна унифицированная модульная среда исполнения, основанная на унифицированной системе типов. Пожалуй, именно эта часть интенсивного программирования проработана наиболее глубоко, см. мой доклад[8] на Открытой конференции ИСП РАН им. В.П. Иванникова в 2021 году.

Спецификация компонент

Спецификация нужна как «человеческая» – документация, так и формальная. Тут все достаточно очевидно, в том числе и проблемы, которые возникнут в реализации.

Возможность гибкой настройки или подстройки

Это очень важное требование, но его трудно описать в общем. Опишем возможные виды компонент и настройки для каждого вида.

Виды компонент:

1. Компонента-функция. Это функция в обычном понимании, которую можно вызвать из другого кода. Очевидный пример, математическая функция, например, sin. Очевидна полезность настройки для такой компоненты, например, точность. Или выбор между скоростью вычисления и использованной памятью. Замечу, что sin скорее всего имеет множество реализаций. На каких-то платформах (и при подходящих требованиях) лучше использовать аппаратную реализацию, для каких-то используется специфическая реализация, а для каких-то самая общая.
2. Компонента-класс. Это тоже понятно, типичный пример — контейнер (Map, Set, Tree, …). Очевидно, нужна настройка на тип (типы) хранимых данных. Столь же очевидно полезна настройка: скорость/память.
3. Конкретизируемые компоненты, или многопараметрические компоненты, которые должны быть адаптированы к особым условиям ее применения (используется терминология А.П. Ершова [Ершов, 1994]). Очевидные примеры: СУБД должна быть адаптирована (настроена) структурой таблиц, а нейросеть должна быть «обучена».
4. Генераторы компонент. Очень близко к конкретизируемым компонентам. Разница только в том, что используется не общий код, как в СУБД, а код, построенный по некому образцу для

6 А. Недоря. Технология разработки мультиплатформенных программ на основе явных схем программ. http://digital-economy.ru/stati/tekhnologiya-razrabotki-multiplatformennykh-programm-na-osnove-yavnykh-skhem-programm

7 А. Недоря. Триада языков программирования. http://xn--80aicaaxfgwmwf3q.xn--p1ai/?p=298

8 А. Недоря. Разработка типовой системы языка программирования приложений. http://алексейнедоря.рф/?p=394

конкретных настроек. Пример: конечные автоматы, генераторы компиляторов и т.д. Другой пример, реализация обобщенных типов данных (списки, деревья, отображения) может быть основана на конкретизации или генерации (репликации) — как мы знаем, в разных языках программирования используются разные подходы.

5. Схемные компоненты [6]. Если компоненты 1-го и 2-го вида относились к синтезирующему программированию (в терминологии Ершова), а следующие компоненты — к конкретизирующему программированию, то здесь мы переходим к сборочному программированию. А именно, к возможности сборки программы из компонент, которые представляют собой черные или серые ящики. Типичный пример — это виджеты, используемые при сборке интерфейса в декларативных UI DSL. Для таких компонент важно разделение на компоненты и схему — компоненты взаимодействует с другими компонентами в соответствии со схемой, внешней по отношению к компонентам. Для схемных компонент настройка задается схемой. Это должно быть понятно по аналогии с декларативным UI. Компонента «Текст» настраивается текстом, а компонента «Кнопка» передает нажатие другой компоненте, и это задается схемой.
6. Компонента — узел. Это схемная компонента более высокого уровня, состоящая из схемы и набора компонент более низкого уровня. С точки зрения использования — это схемная компонента, различия только во внутреннем устройстве.
7. Компонента – схема программной системы. Рассмотрим распределенное приложение, которое состоит из частей, которые исполняются на разных устройствах. Исполнение может быть максимально распределенным — каждая часть исполняется на отдельном устройстве или частично распределенным, то есть некоторые части работают на одном устройстве. Соответственно, есть максимальная схема и отображение на конкретную конфигурацию. В данном случае переиспользуемым продуктом является сама схема, а настройка её — отображение виртуальных устройств на конкретные, с учетом возможного масштабирования.

В среде Вир мной были полноценно проработаны компоненты вида 1, 2, 5 и 6, и экспериментально компоненты вида 3.

Еще раз обратившись к терминологии А.П. Ершова, замечу, что

- компоненты вида 1, 2 относятся к синтезирующему программированию;
- компоненты вида 3, 4, 7 — к конкретизирующему программированию;
- компоненты вида 4, 5 — к сборочному программированию.

А все вместе близко к тому, что А.П. Ершов называл Лексиконом программирования [9].

Что же даст интенсивное программирование?

Как минимум:

- существенное снижение сложности и ускорение разработки — конкретизация и сборка существенно проще и быстрее синтеза;
- повышение надежности софта — многократно используемые компоненты могут быть верифицированы, или, по меньшей мере, лучше протестированы;
- увеличение применимости математических методов, за счет унификации и возможности тестирования отдельных компонент и, далее системы из верифицированных компонент. Это, безусловно, потребует изменения методов, но потенциально приведет к качественному скачку надежности.
- резкое уменьшение потребностей в «чистых» программистах. Так как разработка программ станет проще, то программирование (как разработка инструментов-помощников) станет скорее добавкой к основной профессии.
- Существенное уменьшение ресурсов, необходимых для разработки, тестирования и хранения исходного текста.
- Переход к единой экосистеме исполнения/разработки — в рамках страны или человечества.

Я уверен, что социальные последствия будут существенны, но это уж совсем не моя тема.

Окно возможностей

На мой взгляд, мы находимся на том этапе развития, на котором возможен переход к интенсивному программированию. Именно возможен, но не предопределен.

Сообщество (корпорация, страна или человечество), которое вложит силы и сможет перейти к интенсивному программированию, получит изрядный бонус к развитию. Per aspera ad astra — вперед к галактической империи.

Почему я пишу об интенсивном программировании сейчас?

На мой взгляд, перед Россией открылось окно возможностей, и именно сейчас переход к интенсивному программированию возможен.

Во-первых, мой опыт в программировании однозначно говорит о том, что для того, чтобы сделать существенный рывок вперед, надо выбросить старый мусор. Уточню, выбрасывать надо не идеи, не

9 См. http://rkka21.ru/ershov-lexicon.htm

понимание и не математику. Все это надо хранить, так же, как и людей, которые умеют. Отбрасывать надо «гигатонны» устаревшего кода и инструментов.

Например, в LLVM вложены многие тысячи человеко-лет, и, хотя уже понятно, что через некоторое время этот софт рухнет под собственной тяжестью, никто не начинает делать нечто лучшее — слишком долго и дорого. Но, предположим, что Интел перестал продавать нам свои процессоры. И тогда разработка замены LLVM сразу станет проще, так как можно выкинуть огромное количество трюков, годных только для старых процессоров. Отбрасывание старого может упростить разработку нового в десятки и сотни раз.

Именно сейчас, мы можем отбросить очень многое, и сделать только то, что нужно. Уверен, что это нужное и хорошее будет гораздо меньше, чем то, что есть сейчас.

Во-вторых, переход на интенсивное программирование актуален, в связи с отъездом части программистов из России.

И, в-третьих, у нас есть люди, которые видят путь.

Литература

1. Ершов, А. П. Предварительные соображения о лексиконе программирования// Избранные труды. Новосибирск, 1994.

References in Cyrillics

1. Ershov, A. P. Predvaritel`ny`e soobrazheniya o leksikone programmirovaniya// Izbranny`e trudy`. Novosibirsk, 1994.

Ключевые слова

Интенсивное программирование, сборочное программирование, конкретизирующее программирование

Недоря Алексей Евгеньевич *кандидата физико-математических наук*
(aleksei.nedoria@synergetic-lab.ru)

Alexey Nedorya, Intensive Programming

Keywords

Intensive programming, assembly programming, concretizing programming

DOI: 10.34706/DE-2022-05-12
JEL classification C02 Математические методы

Abstract

The article examines the state of programming through the concepts of "extensive" and "intensive". Based on statistical data, it shows the "extensive" natue of current programming and describes the path of transition to intensive programming and its advantages.

.