ИПМ им.М.В.Келдыша РАН

Абрау-2023 • Труды конференции

Т.А. Андреева, Л.В. Городняя

Можно ли измерять вклад программистских решений в производительность программ?

Рекомендуемая форма библиографической ссылки

Андреева Т.А., Городняя Л.В. Можно ли измерять вклад программистских решений в производительность программ? // Научный сервис в сети Интернет: труды XXV Всероссийской научной конференции (18-21 сентября 2023 г., онлайн). — М.: ИПМ им. М.В.Келдыша, 2023. — С. 12-24.
https://doi.org/10.20948/abrau-2023-2
https://keldysh.ru/abrau/2023/theses/2.pdf

Видеозапись выступления

Можно ли измерять вклад программистских решений в производительность программ?

Т.А. Андреева[1, 2], Л.В. Городняя[1, 2]

[1] Институт систем информатики им. А.П. Ершова (ИСИ СО РАН)
[2] Новосибирский государственный университет

Аннотация. Статья посвящена вопросам оценки влияния программистских решений на продуктивность программирования и производительность программ, их связи с процессом обучения программированию и улучшения программ на практике. Не исключено, что функциональные модели могут быть полезны как метрическая шкала, позволяющая отделять особенности используемых языков и систем программирования от характеристик программ. Именно такой подход рассматривается в данной статье.

Ключевые слова: измерение качества программ, продуктивность программирования, производительность программ, программистские решения, функциональное программирование

Can the Contribution of Software Decisions to Program Performance Be Measured?

T.A. Andreyeva[1, 2], L.V. Gorodnyaya[1, 2]

[1] A.P. Ershov Institute of Informatics Systems
[2] Novosibirsk State University

Abstract. The article concerns measurement of the effect that programming solutions have on productivity of programming and on program performance, and to their connection with educational programming and program improvement in practice. When program effectiveness is measured directly, the productivity of a complex consisting of a computer, a compiler, and a program is really measured instead of the productivity of programming solutions. These measurements do not fully reflect the contribution of programming solutions. And the functional models can provide a metric scale capable to separate features of programming languages and systems from features of programs and programming solutions. One of possible approaches is to measure the productivity of programs and programming solutions on the basis of normalized functional forms and thereby to reduce the dependence of measurements on hardware and programming systems.

Keywords: program quality measurement, programming productivity, program performance, programming decisions, functional programming

Введение

В лаборатории информационных систем ИСИ СО РАН традиционно изучаются вопросы, возникающие в связи с проблемой оценки влияния программируемых решений на продуктивность программирования и производительность программ. В данной статье рассмотрены результаты применения довольно известного бенчмарка «Какой язык быстрее всех?»[1], заслужившего внимание программистов-практиков и студентов[2], результаты сравнения производительности языков Lisp, C/C++ и Java[3] и привлечены механизмы платформы JDoodle[4], нацеленной на поддержку обучения и самообучения практическому программированию с помощью доступных онлайн компиляторов для 76 языков программирования (ЯП). Часть наблюдений в этом направлении фактически происходит при оценке учебных и олимпиадных задач по программированию.

В данной статье рассматривается возможность использовать функциональные модели в качестве метрической шкалы, позволяющей отделять особенности используемых языков и систем программирования от характеристик программ и запрограммированных решений.

Изложение начинается с анализа результатов игры «Какой язык быстрее всех?» на примере языков C++, Go, Clisp, Java, Haskell, Pascal, Python и сравнения Lisp, C/C++ и Java. Далее описан многолетний опыт оценки учебных и олимпиадных работ по программированию, проявивший ряд не вполне очевидных аспектов проблемы. Затем приведены результаты компьютерного эксперимента по измерению производительности этих ЯП и языка Closure на платформе JDoodle на примерах программ решения задачи вычисления чисел Фибоначчи, опубликованных на сайте «Энциклопедия языков программирования»[5] [1].

1. «Какой язык быстрее всех?»

Организаторы эксперимента по проверке скорости ЯП сразу предупреждают: «Будем реалистами: большинство людей, как правило,

---

[1] Gouy, Isaac. The Computer Language Benchmarks Game “Which programming language is fastest?” — https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[2] Некоторые студенты включаются в соревнование за создание более быстрых решений эталонных задач.

[3] Sachin, Patil. Lisp as an Alternative to Java. — https://psachin.gitlab.io/lisp-java-notes.html

[4] Платформа JDoodle. — https://docs.jdoodle.com

[5] Энциклопедия языков программирования. — http://progopedia.ru

ничуть не озабочены производительностью программ» («It's important to be realistic: most people don't care about program performance most of the time»). Чтобы узнать, какой ЯП (точнее, его компилятор) самый быстрый, они решили измерить их в стиле спортивного чемпионата. Проект развивался с 2002 по 2021 год, и в отчёте о его работе представлены данные об испытаниях заметного числа языков. Измеряли скорость, память и объём программы.

У организаторов эксперимента возникла гипотеза, что можно собрать коллекцию эталонных задач и написать программы их решения на разных языках. Обнаружение подобия с эталонами означает существование неявной общей модели, позволяющей анализировать программы на соответствие ожиданиям пользователя, т.е. на решение проблемы применимости (usability) [2]. Большинство компиляторов выполняют преобразования семантики программ, сохраняющие их функциональную эквивалентность. В зависимости от состава таких преобразований, полученный в результате компиляции код программы может различаться по производительности. Организаторы эксперимента столкнулись с рядом проблем, решить которые удалось лишь частично.

Первая проблема: нет критериев выбора эталонных задач.

Вторая проблема: как сравнивать по скорости слишком разные ЯП?

Третья проблема: установление эквивалентности программ. Функционально эквивалентные программы могут не обладать семантической эквивалентностью.

Четвёртая проблема: не все важные особенности программ вообще допускают непосредственное измерение.

Учитывая ещё больший комплекс проблем, организаторы эксперимента выбрали нечто среднее между хаосом и жёсткостью – гибкостью и игрой, чтобы можно было независимо создавать программы на разных ЯП, а не просто механически конвертировать с одного ЯП на другой. Они отдавали себе отчёт в том, что пока нет научных оснований для такого сравнения и не выяснено, насколько измерение производительности программ зависит от производительности компиляторов.

Организаторы эксперимента рассудили, что лучший выбор эталонов для измерения производительности – это реальные приложения[6]. Поэтому организаторы не включали в качестве эталонных задач ядра, представляющие собой небольшие ключевые части реальных приложений, игрушечные программы из начальных заданий по программированию и синтетические тесты, представляющие собой небольшие «поддельные»

[6] Представленные в отчёте программы задачи (n-body, spectral-norm, mandelbrot, pidigits, fasta, k-nucleotide, reverse-complement, binary-trees, simple) не являются «реальными приложениями».

программы. Для большинства задач было выполнено несколько вариантов решений, показавших различную производительность.

Пусть дистанция от учебных и игрушечных программ до реальных приложений велика, она может дать научные основания методике измерений производительности программ.

Для эксперимента была выбрана небольшая коллекция задач[7], к решению которых предъявлены чёткие требования по методу их решения и объёму обрабатываемых данных, гарантирующие функциональную эквивалентность программ решения одной и той же задачи. При этом организаторы эксперимента признали, что за почти 20 лет проекта у них не было времени проверять исходный код созданных приложений, чтобы убедиться, что разные реализации программы сопоставимы, что действительно это одна и та же программа, написанная на разных ЯП. Сравнение программ друг с другом происходило в таком духе, будто разные ЯП были созданы для одной и той же цели, что отнюдь не так.

Организаторы сравнения ЯП признали, что пока не удалось научиться определять относительную производительность ЯП, а лишь накоплен опыт, позволяющий получить простое представление об общей производительности каждого ЯП (точнее, его компилятора) [1, 10]. Видно, что разброс скорости редко превосходит 1 к 10. Python в этом чемпионате часто на последних местах. Это подтверждает исходное предположение о слабом интересе практиков к производительности программ и даже систем программирования, им важнее продуктивность программирования.

В этом плане показательны результаты другого проекта, посвященного сравнению языков Lisp и Java, выполненного 14-ю программистами-добровольцами, написавшими 16 программ (12 на Common Lisp и 4 на Scheme)[8], производительность которых сравнима с C/C++ и Java с точностью до минимизации рисков при выполнении проектов[9]. Результаты эксперимента показали, что быстрейший Clisp или Scheme уступают по скорости работы программ быстрейшим C/C++. При сравнении производительности с C/C++ и Java было констатировано, что Clisp обладает более быстрым циклом отладки программ, чем C/C++ или Java, что означает более высокую продуктивность программирования [3].

Экспериментаторы отметили, что программисты с меньшим опытом программирования на Clisp или Scheme нередко программируют быстрее и

---

[7] Постановки задач, требования к их решению и программы их решения на всех испытуемых ЯП представлены на сайте проекта «Какой язык быстрее всех?» (“Which programming language is fastest?”) — https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[8] В источнике нет намёка на характер задач.

[9] Sachin, Patil. Lisp as an Alternative to Java. — https://psachin.gitlab.io/lisp-java-notes.html

лучше, чем более опытные программисты на C/C++ или Java. Они пришли к выводу, что программирование на языке Lisp имеет тенденцию повышать квалификацию программистов, делать их хорошими программистами на любом ЯП. Не исключено, что язык Lisp даёт программистам больше времени на продумывание и улучшение программ.

Традиционный вопрос, почему столь замечательный язык как Lisp не обладает адекватной популярностью, пока не получил простого ответа. Одна из гипотез: во всем виноват миф о том, что Lisp слишком большой и медленный, существующий одновременно с утверждением, что Lisp лёгкий и элегантный. В противовес этому мифу авторы сравнения языков Lisp и Java приводят результаты своего эксперимента, отмечая характеристики продуктивности программирования и квалификации программистов.

В целом, результаты вышеописанных экспериментов показывают, что проблема измерения производительности программ далека от конструктивного решения. Организаторы этих экспериментов приняли ряд важных решений и констатировали ряд не преодолённых трудностей.

1. Они признали, что нужны эталонные задачи, но не дали пояснений относительно эталонности в условиях непрерывного развития ЯП.
2. Они не дали рекомендаций по сопоставлению реальных приложений с предложенными эталонными, довольно искусственными, задачами.
3. Осталась не выясненной взаимосвязь между продуктивностью программирования и достижением производительности программ. В этом отношении системы обучения программированию и проведения конкурсов по программированию обычно имеют правдоподобные оценки трудоёмкости выполнения учебных и олимпиадных задач.
4. Ещё одна сторона оценки продуктивности и производительности связана с их зависимостью от уровня квалификации, способностей, образования и опыта программистов.

Можно обратить внимание на то, что прогресс технологий программирования обусловлен ростом технологичности, приоритетами которой являются повышение продуктивности программирования и расширение сферы применения программ, что чревато потерями в производительности программ. Нередко происходит и снижение требований к квалификации программистов [4, 5].

2. «Дайте мне точку опоры!…»

Эти слова Архимеда в практическом программировании понимают как «Дайте мне доступ к компилятору с самой маленькой работающей программой, остальное я пойму сам в непосредственном эксперименте». Работающий компилятор для практика является единственно достоверным

документом о реализованном ЯП. Документация – это не более чем справочный материал, нуждающийся в проверке на соответствие тому, что фактически поддерживает компилятор.

Именно такое понимание процесса изучения ЯП лежит в основе платформы JDoodle[10], содержащей онлайн-компиляторы, калькулятор и встроенный редактор, позволяющие осваивать программирование восходящим методом снизу вверх на базе любых из 76-ти ЯП и двух баз данных. Платформа JDoodle не гарантирует точности или надежности предложенных материалов, что не мешает использовать ее в экспериментах и в учебном процессе. Остаются за бортом вопросы решения новых или слабо исследованных задач, характерных для современного программирования в условиях совершенствования аппаратуры и расширения круга пользователей, не обязанных обладать познаниями в сфере теоретического и практического программирования.

3. Опыт оценивания учебных и олимпиадных программ

Автоматизированное оценивание эффективности программ во многом напоминает автоматическое тестирование олимпиадных задач по программированию. Но если первое ещё находится на этапе становления, то второе уже прошло большой путь развития, поэтому для того чтобы иметь возможность позаимствовать удачные решения и не совершить похожих ошибок, сделаем небольшой экскурс в историю олимпиадного тестирования.

На заре школьного олимпиадного движения по программированию[11] олимпиады всех уровней проходили с обязательным включением в них теоретического тура, причём на олимпиадах нижних уровней – школьном и даже районном – практического тура могло вообще не быть. Объяснялось это, конечно же, весьма небольшим количеством имевшихся в школах компьютеров.

Как проходили теоретические туры по программированию? Текст программы или алгоритма участники писали на бумаге, а проверяющие затем читали эти тексты и, выступая в роли синтаксических анализаторов и виртуальных машин, имитировали выполнение программ у себя в голове.

Задания практического тура уже тогда проверялись при помощи тестирования по типу «чёрного ящика» (способ тестирования, когда заключение о внутренних свойствах исследуемого объекта делается лишь на основании его откликов на различные раздражители). Организаторы олимпиады вручную запускали каждую программу с заранее

---

[10] Платформа JDoodle. — https://jdoodle.com

[11] Напомним, что предмет «Основы информатики и вычислительной техники» появился в программе российских – тогда ещё советских – общеобразовательных школ в 80х годах прошлого столетия.

подготовленными входными данными и фиксировали результаты выполнения. Даже на олимпиадах городского уровня участникам разрешалось писать программы с чтением данных не из файла, а с клавиатуры, и ввод перекладывали опять же на проверяющих.

Тем не менее, даже в таких условиях основные черты тестирования по типу «чёрного ящика» просматривались уже тогда.

Во-первых, разрешалось пользоваться почти любым известным участнику языком программирования (к счастью, тогда их выбор был довольно ограничен). В число разрешённых входил даже язык РАЯ (Русский Алгоритмический Язык, разработанный под руководством А.П. Ершова специально для первого школьного курса программирования).

Во-вторых, хотя организаторы олимпиады принимали исходные тексты программ, а компиляция и выполнение тестов производились вручную, собственно тексты рассматривались только в исключительных – спорных или непонятных – случаях.

В-третьих, общее время работы каждого теста не фиксировалось – следили лишь за тем, чтобы в процессе выполнения не происходило зацикливания.

И в-четвёртых, вывод о правильности или неправильности решения и реализующей его программы строился на основании результатов выполнения единых для всех участников тестов.

Автоматизация процесса проверки олимпиадных решений началась практически одновременно с возникновением программистских олимпиад, была, разумеется, постепенной и на каждом этапе использовала все имевшиеся на тот момент возможности вычислительной техники. Однако от принципа проверки по типу «чёрного ящика» не отступали никогда. Результат мог выглядеть двояко: «*Решение прошло все тесты из тестового набора – участник получил оговорённое количество баллов, иначе получил 0 баллов»* либо «*Решение прошло только некоторые тесты из тестового набора – участник получил не максимальное, но и не нулевое количество баллов»*. Конечно же, во втором случае стоимость «тяжёлых» тестов была заметно выше и участники, выбравшие более эффективные алгоритмы решения, получали преимущество. Эти два подхода к оцениванию прекрасно уживаются и по сей день.

Когда развитие компьютерных сетей и операционных систем сделало возможным автоматизировать сбор решений, их компиляцию и выполнение, появились некоторые ограничения, обусловленные особенностями тестирующих систем: только файловые ввод и вывод данных, использование только «одобренных» компиляторов и т.п. – и дополнительный критерий: эффективность выбранного алгоритма. Алгоритм решения не должен приводить к зацикливанию, переполнению памяти, критическим ошибкам типа «деление на ноль» и т.п. даже в редких

частных случаях, а также должен уметь справляться с большими объёмами данных, укладываясь при этом в заданный временной отрезок. Например, для того чтобы «отсеять» решения, использовавшие не алгоритм Дейкстры (его сложность, напомним, пропорциональна N^2, где N – количество вершин в графе), а рекурсивный полный перебор (сложность которого пропорциональна 2^N), в тестовый набор вводились тесты с большим количеством вершин, что заведомо выводило переборный вариант за временные ограничения.

Автоматический подсчёт количества времени, затраченного на выполнение программы, ведётся и в системе сравнения эффективности программ. Но здесь на первый план выходит ещё одна тонкость: применение оптимизирующих компиляторов. На многих олимпиадах жёстко оговаривается не только набор разрешённых компиляторов, но и набор разрешённых опций компилирования. Одна из таких спорных опций – оптимизация. Поскольку её результат довольно сильно зависит от конкретной реализации выбранного участником алгоритма, наиболее правильным подходом, уравнивающим всех участников, является отключение этой опции: она может исказить результаты измерений как в ту, так и в другую сторону.

Другим результатом автоматизации тестирования стало исчезновение семантического анализа представленных решений. Если человек-проверяющий может понять смысл программы, то автомату это недоступно. Тестирование олимпиадных программ свелось только к проверке их работоспособности на некотором тестовом наборе. «Чёрный ящик» победил. И если в первые годы школьного олимпиадного движения вполне была возможной ситуация «*Участник взял эффективный алгоритм, но в его реализации допустил ошибку, – за это мы добавим ему поощрительные баллы, хотя его решение и не прошло ни одного теста*», то при полностью автоматическом тестировании учитывается только факт успешного прохождения тестов.

Но является ли программа, успешно прошедшая все тесты из тестового набора, правильным решением поставленной задачи? Разумеется, это напрямую зависит от тестового набора. Например, при отсутствии в тестовом наборе тестов, описывающих какой-либо редко встречающийся частный случай, нельзя утверждать, что программа является правильной. Она лишь правильно работает на некоторых наборах входных данных.

На многих олимпиадах в конце соревнования набор проверочных тестов становится доступным всем участникам и на основании анализа этого тестового набора даже принимаются апелляции. Но в системе

бенчмарка «Какой язык быстрее всех?»[12] набор тестов, на основании которых делается вывод о эффективности или неэффективности предложенной программы, скрыт, что делает не очень достоверными результаты проверки. Чаще всего «камнем преткновения» в вопросе эффективности выбранного языка программирования, алгоритма и его реализации являются именно частные или граничные случаи. Понятно, что при $N = 10$ разница в эффективности упомянутых ранее алгоритма Дейкстры и рекурсивного алгоритма полного перебора будет сравнима с ошибкой измерений. И совсем другая картина сложится при $N = 10\,000$. Таким образом, невозможно выстроить достоверную систему сравнения эффективности программ без всестороннего анализа области допустимых входных данных.

Как видим, автоматическое тестирование решений задач по программированию и измерение эффективности реализаций различных алгоритмов на различных языках программирования действительно имеют много общего. Хочется, чтобы измерение эффективности в своём дальнейшем развитии учло и «обезвредило» те особенности автоматического тестирования, которые способны исказить результаты измерений.

Подготовка и проведение лабораторных работ и олимпиад по программированию включает в себя ряд уже зарекомендовавших себя средств и методов комплектации и проверки заданий и отдельно оценивания и тестирования программ решения задач [6, 7, 8, 9]. Уже достигнуто понимание ряда особенностей выбора и требования к постановкам задач, известны типичные категории задач, упорядоченные по сложности.

Первые эксперименты можно продолжить на числовых функциях, легко масштабируемых одним параметром: факториал, числа Фибоначчи, простые числа, степенные ряды и т.п. [4]. Можно задействовать вещественные и мнимые числа в форме решения квадратных уравнений, извлечения корней, дробных формул с риском деления на ноль. Можно использовать символьную обработку, перестановки или разрастающиеся цепочки [5].

Шкала сложности программируемых решений может различать

- на первом уровне: элементарные операции, простые переменные, ветвления, доступ к соседнему элементу последовательности, строки, векторы, списки и стеки;
- на втором уровне: функции и процедуры, иерархию областей действия, разные виды ветвлений, структуры данных;

---

[12] Gouy, Isaac. The Computer Language Benchmarks Game “Which programming language is fastest?” — https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

– на третьем уровне: ограничения, диагностики, ловушки, контроль типов данных, рекурсию и циклы;
– на четвёртом уровне: отображения, преобразования, фильтры, свёртки, генераторы и конструкторы, макросы;
– на пятом уровне: многопоточность и имитацию обменов данными между потоками без сложных взаимодействий.

Важно, что связывание производительности программ с программируемыми решениями при таком подходе допускает сопоставление со спектром используемых средств языка или эталонными задачами. Разброс производительности программ, кроме того, обусловлен продуктивностью программирования, т.е. квалификацией программистов, владеющих не только методами решения таких задач, но и определёнными способностями, средствами, техникой программирования. Всё это дополняется умением самостоятельно решать проблемы в пространстве, выходящем за пределы языка и системы программирования, что более подробно описано в работах [10, 11], представляющих парадигмальную методику анализа и сравнения языков программирования, а также неформализуемые особенности процесса разработки программ.

Конечно, полное изложение почти сорокалетнего опыта олимпиадного, спортивного программирования заслуживает отдельной статьи, здесь приведена лишь краткая справка, связанная с продуктивностью программирования и олимпиадный опыт оценки программ через тестирование с учётом времени решения задач в комплекте. Это показывает проблему измерения качества программ с точки зрения принятия решений с одной стороны участниками, с другой стороны организаторами программистских чемпионатов. Отражает ли интегральная оценка продуктивность работы участников и производительность сделанных ими программ? Что из опыта оценки олимпиадных работ может получить развитие как методика оценки учебного и производственного программирования или что обладает сходством, а что резко отличается?

4. Эксперимент по измерению производительности программ

Результаты предварительного эксперимента на базе платформы JDoodle[13], выполненного на примере вычисления чисел Фибоначчи для «зондирования почвы», показывают возможность измерения вклада программируемых решений. Этот алгоритм интересен тем, что даёт быструю потерю производительности.

Семантическая эквивалентность измеренных программ позволяет разницу в производительности относить на счёт разницы в компиляторах и аппаратуре, когда запрограммированные решения одинаковы. Варианты

[13] Платформа JDoodle. — https://jdoodle.com

программируемых решений одного и того же метода могут использовать разные средства ЯП и потому быть семантически не эквивалентными.

Разница в показателях Java и Clojure, вероятно, объясняется различием в границах счёта. Это не удивительно, ЯП создаются для разных целей. Программа на языке Java допускает наибольший диапазон данных. Язык C++ работает столь быстро, что обработку малых объёмов там трудно замерить. Язык Clisp поддерживает заботу о малых заданиях, даже давая им приоритет. Похожий на него Clojure – лидер на больших объёмах и активный «пожиратель памяти». Компилятор языка Haskell во многом компенсирует недостатки языка. Go и Java близки по ряду показателей. И конечно, Pascal показывает, что для успешного решения учебных задач многого не надо. Разница в производительности таких вариантов 1 к 1000 уже объясняется программируемыми решениями. Разброс показателей намного превышает разницу между прогонами на разных компиляторах.

Подобную разницу можно видеть и по результатам «игры» в самый быстрый компилятор, но, как правило, с заметно меньшим разбросом. Беглый взгляд на самую быструю и самую медленную программы, написанные на языке Clisp одним и тем же программистом (John Smith), показывает, что заметная разница в скорости скорее всего объясняется использованием в быстрой программе макросов и многопоточности. Разница между вариантами функционально эквивалентных программ, не обладающих семантической эквивалентностью, на одном и том же ЯП для приведённых примеров существенно выше, чем разница между семантически эквивалентными программами на разных ЯП. Впрочем, пока рассмотрено слишком мало примеров и ЯП для уверенных выводов.

Первые кандидаты в эталонные языки – это языки функционального программирования, обладающие высокой моделирующей силой. На этапе экспериментов достаточно выделять из программ ключевые фрагменты, представляющие программируемые решения, приводить их к нормализованной форме на эталонном языке для измерения.

К проведению эксперимента привлечены слушатели ежегодного спецкурса «Парадигмы программирования», читаемого в НГУ на факультетах ММФ и ФИТ.

Заключение

В данной статье описаны результаты двух подходов к оценке производительности программ и удобный бенчмарк для постановки обучения программированию, нацеленного на формирование профессионального умения оценивать и измерять вклад программируемых решений в производительность программ. Изложение опыта оценивания решений олимпиадных задач по программированию показывает перспективу формирования научно обоснованной методики измерения

производительности программ. Приведены результаты небольшого эксперимента, дающего основания для функционального подхода к разработке методики измерения качества программ.

Основные выводы и гипотезы:

1. Исследование и разработку метрик производительности программ необходимо проводить одновременно с созданием методик прогнозирования продуктивности разработки программ.
2. Задачи можно измерять в терминах задач, а программы в терминах программ, используя подобие.
3. Первые эксперименты можно выполнить на материале учебных и олимпиадных задач, характеристики которых хорошо известны.

Усложняющим является требование понятности метрики как для программистов, так и для пользователей и менеджеров, но без этого метрика не может получить признание. В дальнейшем предстоит экспериментальное исследование предложенной методики на более широком наборе задач, языков программирования и особенностей их разработки и применения [11].

Литература

1. Андреева Т.А., Городняя Л.В. Функциональный подход к измерению вклада программируемых решений в производительность программ : препринт. — Новосибирск, 2022. — 62 с. — https://www.iis.nsk.su/files/preprints/preprint_187.pdf
2. Купер А. Психбольница в руках пациентов. Алан Купер об интерфейсах. — СПб.: Питер, 2018. — 384 с.
3. Sachin, Patil. Lisp as an Alternative to Java. — https://psachin.gitlab.io/lisp-java-notes.html
4. Weinberg G.M. The Psychology of Computer Programming. — Silver Anniversary Edition, 2011. — 288 p.
5. Липаев В.В. Человеческие факторы в программной инженерии. Рекомендации и требования к профессиональной квалификации специалистов. — М.: СИНТЕГ, 2009. — 348 с.
6. Андреева Т.А. Возможность автоматизации процесса генерирования тестовых наборов // Universum: технические науки. — №8(41). — М., Изд. «МЦНО», 2017. — С. 5–7.
7. Andreyeva, T.A. Automation of correctness checking in education // A.P. Ershov Informatics Conference / Educational Informatics Workshop proceedings. July 2–3, 2019. — Novosibirsk, 2019. — P. 6–15.
8. Андреева Т.А. Сборник задач для предолимпиадной подготовки по программированию. — Новосибирск: Изд-во НГУ, 2009. — 226 с.

9. Андреева Т.А. Программирование на языке Pascal. — М.: ИНТУИТ, 2016. — 277 с.
10. Gouy, Isaac. The Computer Language Benchmarks Game “Which programming language is fastest?” — https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
11. Городняя Л.В. Функциональное программирование. Парадигма, модели, методы. — Новосибирск: Изд-во СО РАН, 2022. — 482 с.

References

1. Andreyeva T.A., Gorodnyaya L.V. Funktsionalnyj podhod k izmereniju vklada programmiruemyh reshenij v proizvoditelnost programm : preprint. — Novosibirsk, 2022. — 62 p. — https://www.iis.nsk.su/files/preprints/preprint_187.pdf
2. Kuper A. Psihbolnitsa v rukah patsientov. Alan Kuper ob interfejsah. — SPb.: Piter, 2018. — 384 p.
3. Sachin, Patil. Lisp as an Alternative to Java. — https://psachin.gitlab.io/lisp-java-notes.html.
4. Weinberg G.M. The Psychology of Computer Programming. — Silver Anniversary Edition, 2011. — 288 p.
5. Lipaev V.V. Chelovecheskije faktory v programmnoj inzhenerii. Rekomendatsii i trebovanija k professionalnoj kvalifikatsii spetsialistov. — M.: SINTEG, 2009. — 348 p.
6. Andreyeva T.A. Vozmozhnost avtomatizatsii processa generirovanija testovyh naborov // Universum: tehnicheskie nauki. — № 8(41). — M., 2017. — P. 5–7.
7. Andreyeva, T.A. Automation of correctness checking in education // A.P. Ershov Informatics Conference / Educational Informatics Workshop proceedings. July 2–3, 2019. — Novosibirsk, 2019. — P. 6–15.
8. Andreyeva T.A. Sbornik zadach dlia predolimpiadnoj podgotovki po programmirovaniju. — Novosibirsk: Izd-vo NGU, 2009. — 226 p.
9. Andreyeva T.A. Programmirovanije na jazyke Pascal. — M.: INTUIT, 2016. — 277 p.
10. Gouy, Isaac. The Computer Language Benchmarks Game “Which programming language is fastest?” — https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.
11. Gorodnyaya L.V. Funktsionalnoe programmirovanije. Paradigma, modeli, metody. — Novosibirsk: Izd-vo SO RAN, 2022. — 482 p.