Харлан Миллз

Структурное программирование: взгляд в прошлое и в будущее

Harlan Mills (1986) Structured Programming: Retrospect and Prospect // IEEE Software, Vol.3, No.6, p.58-66.

1995, А. Л. Китаев, перевод с англ.

В статье анализируются основы структурного программирования и история возникновения этого направления. На примере двух крупных проектов (New York Times и NASA Skylab) делается вывод о высокой эффективности и большом потенциале структурного программирования. Будущее развития этого направления автор видит прежде всего во внедрении доказательного программирования, основанного на идеях математического доказательства корректности создаваемых программ. Отказ от ставшей уже традиционной покомпонентной отладки и тестирования и переход на рельсы функциональной верификации поможет, по мнению автора, добиться гораздо более высокого качества и необходимого уровня надежности сложных программных систем.

С тех пор, как два десятилетия назад появилось структурное программирование, программы стали писать иначе. Однако идеи, предложенные тогда, и в наши дни обладают мощным потенциалом, сулящим большие перемены в разработке программного обеспечения.

В 1969 г. Эдсгер Дейкстра (Edsger W. Dijkstra) опубликовал статью "Структурное программирование" [1], и это событие ознаменовало начало десятилетия напряженного труда над методами программирования, который в корне изменил наши представления о тех возможностях, что открыты человеку в сфере разработки программного обеспечения.

До этого знаменательного десятилетия программирование считалось занятием частным, чем-то вроде решения головоломок, и сводилось оно к написанию компьютерных команд, которые должны были работать как одна программа. Но десятью годами спустя программирование уже можно было рассматривать как деятельность по преобразованию спецификаций в программы, как деятельность, которая обладает общественным значением и которая основана на математических методах.

Раньше проблема состояла в том, чтобы получить хоть как-то работающую программу, а нужных результатов программисты добивались от нее только после проведения длительной отладки. Потом уже никого не удивляло, что программы и выполняются, и делают то, что надо, причем при самой минимальной отладке или даже вообще без нее. Если раньше было принято считать, что ни одна большая программа не может быть свободной от ошибок, то потом появилось множество крупных программ, которые работали годами, не выявив при этом ни одной ошибки.

Влияние структурного программирования. Все эти ожидания и достижения не стали явлением повсеместным: что ни говори, а промышленность — система инерционная. Но они достаточно утвердились, чтобы стать провозвестниками коренных перемен в разработке программного обеспечения.

Как известно, доводы Дейкстры в пользу структурного программирования первоначально состояли в том, что упрощение логики управления позволит в свою очередь упростить доказательство корректности программ. Однако многие и по сей день считают верификацию программ уделом университетских мудрецов — по крайней мере, до тех пор пока автоматические системы верификации не станут настолько быстродействующими и гибкими, чтобы их можно было использовать практически.

Между тем, доводы Дейкстры, утверждавшего, что результаты проведенной человеком неформальной верификации могут быть достаточно надежными и что такую верификацию можно применять вместо традиционной отладки до тестирования системы, подкрепляются эмпирическими данными [2]. Фактически структурное программирование, включающее верификацию корректности программистом, может стать основой разработки программного обеспечения при статистическом контроле качества (statistical quality control) [3].

Человеку свойственно ошибаться, в том числе и при разработке программного обеспечения, но вытекающие отсюда опасности, похоже, сильно преувеличиваются. Появление структурного программирования и устранение большей части ненужных сложностей может открыть перед нами полосу новых надежд и новых достижений.

Первые споры. В своей статье Дейкстра предложил ограничить логику управления программами тремя формами: следование (sequence), выбор (selection) и цикл (iteration). Таким образом, в языках Algol и PL/I оператор "goto" был уже попросту не нужен. А ведь прежде этот оператор был, можно сказать, основой, на которой строились все вычисления с хранимой программой, а главным инструментом мастера в творческом процессе программирования считалась возможность произвольно, в зависимости от состояния данных передавать управление на те или иные участки кода. Разумеется, в операторы выбора и цикла был неявно встроен условный переход, но они казались бледной тенью оператора "goto", предоставлявшего гораздо более широкие возможности.

Итак, предложение Дейкстры запретить оператор "goto" было встречено бурей протестов: "Это же розыгрыш! Да он просто смеется над нами!". Ведь когда перед программистами возникали тяжелые проблемы, они применяли настолько сложные структуры управления, что в их программах — настоящих клубках "компьютерного спагетти" — операторы следования, выбора и цикла, казалось, ни за что не смогли бы выразить всю необходимую логику. Неудивительно, что программисты-практики отнеслись к новой идее скептически: "Может быть, она и подойдет для решения простых задач. Но для сложных задач у нее нет никаких шансов!".

На деле же предложение Дейкстры выходило далеко за рамки простого ограничения управляющих структур. В своих "Заметках по структурному программированию" [4] (которые были опубликованы в 1972 г., но еще в 1970 г. или даже несколько раньше ходили по рукам в виде рукописи) он рассматривал программирование комплексно, как процесс, предусматривающий пошаговое уточнение (stepwise refinement), нисходящую разработку (top-down development) и верификацию программ (program verification).

Однако теоретическую правильность предложения Дейкстры можно продемонстрировать на основе выводов, которые сделали ранее Коррадо Боэм (Corrado Boehm) и Джузеппе Якопини (Giuseppe Jacopini) [5]. Они показали, что логику управления любой программы, состоящей из блок-схем — любой, если угодно, "тарелки спагетти" — можно выразить без пресловутых "goto", с помощью одних операторов следования, выбора и цикла.

Итак, комбинация из трех основных операторов оказалась инструментом куда более мощным, чем это предполагалось, и достаточно мощным, чтобы "укротить" любую состоящую из блок-схем программу. Да, это стало большим сюрпризом для рядовых программистов.

Но как бы там ни было, против предложения Дейкстры было выдвинуто новое возражение: "Данная идея не подходит для решения практических задач". Что ж, психологически реакция объяснимая — надо же было хоть как-то отстаивать правомерность отлаженного к тому времени производства "компьютерного спагетти". Прошло несколько конференций, где основательно обсуждались такие вопросы, как практичность, оригинальность программирования, его творческий характер — но от всего этого было больше шума, чем пользы.

Первый опыт промышленного применения

Проект New York Times. Проблему практичности разрешила появившаяся вскоре вслед за этим публикация, посвященная итогам крупного проекта, в котором использовалось структурное программирование. Ее автор Терри Бейкер (F.Terry Baker) рассказал о проекте, реализованном фирмой IBM для газеты New York Times. В ходе этого проекта, который занял два года и был

завершен к середине 1971 г., методом структурного программирования была создана система из примерно 85000 строк кода [6]. Структурное программирование выдержало нелегкий экзамен!

В проекте одновременно использовалось несколько нововведений: организация команды главного программиста (chief-programmer team organization) нисходящая разработка с пошагововым уточнением, иерархическая модульность и функциональная верификация программ. И все это стало возможным благодаря структурному программированию.

Созданная для New York Times система представляла собой оперативную систему хранения и поиска информации для справочной службы редакции. Доступ к ней был организован с более чем ста терминалов — это действительно была передовая для своего времени система. В соответствии с планом разработчики сумели вывести ее на весьма впечатляющие эксплуатационные характеристики: при использовании временной аппаратной конфигурации Modem 40 она фактически достигла производительности, предусмотренной для машины IBM 360/Model 50.

На высокий уровень производительности вышла и бригада проектировщиков IBM. Комплексное исследование, проведенное в рамках компании, показало, что по сравнению с другими проектами подобного объема и сложности производительность труда в этом случае была в пять раз выше.

Поскольку New York Times не обладала достаточным опытом в организации работы и сопровождения сложной интерактивной системы, IBM согласилась сопровождать систему в течение первого года эксплуатации. В результате были получены точные данные об опыте эксплуатации системы. Эта информация была также опубликована Бейкером [7].

Еще одним приятным сюрпризом была надежность системы. В то время когда другие интерактивные системы программного обеспечения, как правило, давали сбои по несколько раз на дню, разработанная для New York Times система за весь первый год эксплуатации вышла из строя только однажды.

За прошедший год по разным причинам в нее пришлось внести 25 изменений; большая часть из них пришлась на подсистему редактирования данных (data editing subsystem), которая была спроектирована и включена в систему уже после начала реализации проекта. Причем порядка трети упомянутых изменений касались внешних спецификаций, треть приходилась на явные ошибки, и треть можно было отнести как к первой, так и ко второй категории. Вообще же на тысячу строк кода системы для New York Times приходилось лишь 0,1 ошибки. Этот проект, ставший на то время самым высококачественным из всех проектов IBM такого уровня сложности и объема, оказал значительное влияние на практику проектирования программного обеспечения в этой фирме.

Теорема о структурном решении (the structure theorem) и нисходящий подход к разработке. Итак, сомневающиеся получили доказательства того, что структурное программирование в принципе возможно и что оно помогает добиваться высоких результатов при решении практических проблем. Но все же предстоит пройти еще долгий путь, прежде чем оно будет повсеместно принято в крупных организациях и начнет приносить при этом ощутимую пользу. А прийти к этому можно лишь тогда, когда работники предприятий, начиная с верхних эшелонов руководства, получат достаточное представление о структурном программировании и поймут, что его применение позволит им эффективнее решать свои проблемы. Увещевания и призывы вряд ли смогут принести в этом деле какую-то пользу.

На администраторов выводы Боэма и Якопини производят наибольшее впечатление в тех случаях, когда преподносятся в виде так называемой "теоремы о структурном решении" [8]. Эта теорема доказывает возможность создания структурированной программы для любой проблемы, которая допускает решение с помощью разложения на блок-схемы.

Поясним это на примере. Разработчики аппаратного обеспечения неявно пользуются (и, надо сказать, с успехом) аппаратом булевой алгебры и логики с тем, например, результатом, что из элементов НЕ, И и ИЛИ может быть создана любая электрическая цепь. И если какой-нибудь инженер упорствует в том, что таких элементов для этого недостаточно, то тем самым он ставит под сомнение свою квалификацию инженера.

Теорема о структурном решении позволяет организовывать работу, устанавливая стандарты проектирования с учетом особых случаев. Программист не имеет права голословно утверждать, будто та или иная проблема слишком сложна, чтобы решать ее с помощью структурного программирования. В качестве доказательства он должен представить такую программу. Обычно к тому времени, когда программу приносят, проблему представляют себе уже гораздо лучше, чем прежде, и хорошее решение уже найдено. Иногда случается и так, что конечное решение не является структурным — но оно должно быть хорошо документировано и верифицировано в виде особого случая.

Строки текста в структурированной программе можно писать в любом порядке. Как писались строки и каким образом они группировались в окончательную структурную программу — при выполнении программы все это несущественно. Однако если учесть, что работу в конце концов выполняют люди (со всеми их достоинствами и недостатками), то порядок, в котором пишутся строки структурированной программы, может иметь большое значение с точки зрения корректности программы и ее полноты. Так, строки, содержащие команду "открыть файл", следует писать до тех строк, где идут команды "прочитать" или "записать что-то в данный файл". Это позволяет проверить состояние файла при кодировании операторов чтения/записи.

Главная организационная выгода, которую дает нисходящее программирование, была описана в нисходящем подходе (the top-down corollary) [8] для теоремы о структурном решении. В хронологическом плане строки структурированной программы можно писать таким образом, что каждая из них верифицируется лишь относительно уже написанных строк и не привязывается к тем строкам, которые еще предстоит написать.

В отличие от программ-спагетти, структурированная программа выстраивает команды в естественную иерархию, причем эти команды с помощью конструкций следования, выбора и цикла неоднократно вкладываются во все большие части программы. Каждая такая часть программы определяет иерархию более низкого порядка, которая выполняется независимо от своего окружения в данной иерархии. Любую такую часть можно назвать программной заготовкой (program stub) и дать ей имя, но, что куда более важно, ее можно описать в спецификации, которая не имеет управляющих свойств и обладает лишь воздействием программной заготовки на данные самой программы.

Концепция нисходящего программирования, описанная в 1971 г. [9], позволяет с помощью этой иерархии структурированной программы и с помощью программных заготовок, а также их спецификаций разложить проектирование программы на ряд элементов, превратив ее тем самым в иерархию более мелких, независимых проблем проектирования. Одновременно с этим схожую концепцию пошагового уточнения описал Никлаус Вирт (Niklaus Wirth) [10].

Применение нисходящего подхода. В начале 70-х годов нисходящий подход не воспринимался, поскольку программирование повсеместно считалось процессом синтеза инструкций в единую программу, а не процессом анализа, в ходе которого спецификации реструктурируются в конечную программу. Более того, порядок, в котором следует писать строки текста, противоречил практике, широко распространенной в то время в среде программистов.

Например, в соответствии с нисходящим подходом необходимо было сначала воспользоваться языком управления заданиями JCL (job-control language), затем языком управления компоновкой LEL (linkage-control language), а обычные программы на языке программирования надо было писать в самую последнюю очередь. Между тем, в соответствии с общепринятой процедурой все следовало делать как раз наоборот. Далее, постоянные внутренние циклы, с которых обычно и начинали разработку, при работе в соответствии с нисходящим подходом нужно было писать в последний момент. В сущности, нисходящий подход навязывал представление о том, что компоновщик следует рассматривать скорее как транслятор, нежели как обычную утилиту.

Чтобы правильно понять нисходящий подход, нужно иметь в виду следующее обстоятельство: из него отнюдь не вытекает, что мыслить тоже надо в нисходящем направлении. Польза от его применения проявляется на более поздних фазах проектирования программы, когда завершены уже и "восходящее обдумывание" и, возможно, часть пробного кодирования. Затем, уже зная, куда ведет нисходящая разработка, строки структурированной программы можно проверять одну за другой, по мере того как они выходят из-под пера, причем нет необходимости писать дополнительные строки для того, чтобы исправить предыдущие. В крупных проектах этот нисходящий процесс должен давать возможность просчитывать вперед на несколько уровней иерархии, хотя и не обязательно до самого конца.

Разработчики системы для New York Times применяли как теорему о структурном решении, так и соответствующий ей нисходящий подход. И хотя доказательство самой теоремы (основанной на теореме Боэма и Якопини) представлялось более сложным для понимания, специалисты этой группы считали, что применение нисходящего подхода в проектировании программ было более сложным, но и, соответственно, давало более ощутимые результаты.

Так, примерно половина всех модулей, подготовленных для реализации в проекте Times, оказалась корректной уже после их первой чистой компиляции, хотя для этого не пришлось ни прикладывать особых усилий, ни ставить какие-то предварительные цели. Добиться такого результата помогли и некоторые другие приемы, включая организацию бригады главного программиста (chief-programmer team), понятные библиотечные процедуры разработки программ и интенсивная вычитка программ (program reading). Однако, использование всех этих приемов стало возможным в значительной степени благодаря программам, которые были структурированы в соответствии с нисходящим принципом, и в особенности — благодаря способности откладывать и перепоручать задачи по проектированию за счет спецификаций программных заготовок.

Skylab, проект NASA. Эти преимущества нисходящего структурного программирования были продемонстрированы и в ходе гораздо большего по своим масштабам, но получившего значительно меньшую огласку программного проекта. Речь идет о разработке программного обеспечения для системы космической лаборатории Skylab, выполненной фирмой IBM по заказу NASA в 1971-74 годах. Отметим для сравнения, что работы по проектированию системы Apollo для NASA (с ее помощью люди несколько раз побывали на Луне) проводились в 1958-71 годах, то есть начались до того, как концепция структурного программирования была изложена публично.

Если в проект разработки системы для New York Times, реализованный за два с лишним года, была вовлечена небольшая бригада специалистов (сначала это было четыре человека, а потом их число выросло до десяти), то в работах по каждому из проектов Apollo и Skylab участвовало по 400 человек в течение трех лет подряд. В каждой системе программное обеспечение разделялось на две основные части: (1) тренажерная система для подготовки диспетчеров службы управления полетом и астронавтов и (2) рабочая система управления космическим кораблем в ходе полета.

В сущности, каждая из этих подсистем во многих отношениях представляет собой зеркальное отражение другой. Так, тренажерная система оценивает реакцию космического корабля на включение ракетного двигателя для астронавтов, проходящих тренировку, а рабочая система отслеживает реакцию космического корабля на включение ракетного двигателя астронавтами в ходе полета.

Skylab, проект пилотируемого полета для изучения околоземного пространства, не был таким эффектным, как проект Apollo, но во многих отношениях он был более сложным. Программное обеспечение тренажерной системы Skylab примерно в два раза превосходило соответствующее программное обеспечение Apollo по объему, а по уровню сложности разрыв был еще большим.

Разработка программного обеспечения для Skylab началась вскоре после появления первых публикаций, посвященных структурному программированию, то есть как раз в то время, когда возникла реальная возможность провести методологическое сравнение. При проектировании рабочей системы для Skylab были задействованы все те же успешные методы, которые использовались в случае с обеими подсистемами проекта Apollo. Но тренажерная система Skylab'а по инициативе Сэма Джеймса (Sam E.James) разрабатывалась с помощью тогда еще нового метода нисходящего структурного программирования.

Результаты, полученные в ходе выполнения проекта Skylab, развеяли последние сомнения. В процессе работы по проекту Apollo производительность программистов, проектировавших как тренажерную, так и рабочую системы, была почти одинаковой — чего, собственно, и следовало ожидать. Рабочая система Skylab проектировалась с приблизительно такой же производительностью и с такими же трудностями в плане интеграции, что и обе подсистемы Apollo. В то же время при разработке тренажерной системы Skylab, где применялось нисходящее структурное программирование, была достигнута в три раза более высокая производительность труда, тогда как число проблем, осложнявших интеграцию компонент, резко сократилось.

Пожалуй, самым красноречивым был показатель компьютерного времени, израсходованного на решение проблем интеграции. В большинстве проектов того периода на этапе интеграции расход

компьютерного времени значительно возрастал из-за неожиданно возникавших системных проблем. В случае с тренажерной системой Skylab расход компьютерного времени оставался на одном и том же уровне на протяжении всего этапа интеграции.

Проблемы языка. К этому времени (середина 70-х годов) дебаты о практичности структурного программирования фактически прекратились. Конечно, какое-то количество наиболее консервативных людей убедить так и не удалось, но публичные споры уже затихли.

Несмотря на все это, лишь алголоподобные языки позволяли вести прямое структурное программирование с помощью операторов следования, выбора и цикла непосредственно на этих языках. Серьезной проблемой структурного программирования оставались язык ассемблера, Фортран и Кобол.

Один из подходов при работе с этими языками состоит в том, чтобы проектировать с помощью структурированных форм, а затем, на конечном этапе кодирования, вручную переводить формы на исходный язык. Другой подход: создать языковый препроцессор, позволяющий автоматически переводить конечный код с расширенного языка на исходный. Каждый из этих подходов имеет свои недостатки.

В первом случае от разработчиков требуется очень высокий уровень дисциплины и самоотдачи, пожалуй, даже слишком высокий для большинства коллективов. Очень уж велико искушение использовать средства языка, не совместимые со структурным программированием.

Второй подход навязывает свои правила, заставляет четко следовать дисциплине, но программы, скомпилированные на целевом языке — это плод механического перевода (с искусственными метками и переменными), и их будет трудно читать. Не исключено, что создание препроцессора тоже будет делом и обременительным, и дорогим, так что в процессе отладки может возникнуть искушение прямо изменять механически сгенерированный целевой код, подобно тому как "латают" программы на языке ассемблера. А это, как известно, приводит в последующем к потере интеллектуального контроля.

Описанная ситуация "выбора меньшего из двух зол" и объясняет тот факт, что преимущества структурного программирования довольно медленно становятся достоянием программистов, работающих на языке ассемблера, а также на языках Фортран и Кобол.

Парадоксально, но факт: программирование на языке ассемблера, пожалуй, легче всего адаптируется (с помощью макроассемблеров) к структурному программированию. Так, обе системы проекта Skylab — как тренажерная, так и рабочая — программировались на языке ассемблера, причем на тренажерной системе структурное программирование осуществлялось с помощью макроассемблера.

Чтобы сделать возможным прямое структурное программирование на языках Фортран и Кобол, описание последних было модифицировано, но в большинстве случаев работа с этими языками даже сегодня не приносит всех тех преимуществ, которые может дать структурное программирование.

Нынешний состояние теории и практики

Математическая корректность структурированных программ. Что же нового можно узнать о структурном программировании теперь, когда дебаты уже затихли, а скептики покинули поле боя? Как выяснилось, узнать предстояло еще очень многое, и по большей части такого, чего Дейкстра предсказывал еще в своей первой статье [1].

Первые дискуссии по использованию структурного программирования в промышленности велись главным образом вокруг вопросов об отказе от операторов "goto", о теоретических возможностях программ с ограниченной логикой управления, а также о синтактических и типографических аспектах применения структурированных программ (соглашения об отступах и красивая распечатка, пошаговое уточнение по одной странице за раз).

Эти аспекты позволили программистам ежедневно знакомиться с программами коллег, проводить сквозной контроль и проверку программ, а администраторам — получать представление о разработке программного обеспечения как о процессе пошагового уточнения, который дает возможность все более точно оценивать степень готовности проекта.

То есть, когда о проекте, ведущемся в строгом соответствии с канонами нисходящего структурного программирования, говорили, что он готов на 90%, это означало, что для его завершения предстоит выполнить еще 10% работ (а уж никак не еще 90%!).

Но ведь все эти проблемы: и синтаксис, и типографские тонкости, наглядность, пошаговое уточнение, нисходящая разработка — вообще не упоминались в первой статье Дейкстры по структурному программированию. А его главный аргумент в пользу структурного программирования состоял в том, что в этом случае сокращаются математические доказательства корректности программ! Аргумент, кстати, тем более странный, что о доказательстве корректности своих программ почти никто не беспокоился в то время (да и мало кто беспокоится сейчас). И все же идеи Дейкстры оказались вдохновенным пророчеством, которое сбывается и поныне.

В ходе популяризации структурного программирования речь шла главным образом о его синтактических и внешних аспектах, и происходило это потому, что такие аспекты легче всего поддаются объяснению. Но ими структурное программирование отнюдь не исчерпывается (а коли речь идет о преимуществах, то не исчерпывается и наполовину, ибо существует замечательная взаимосвязь между структурным программированием и математической корректностью программ. И людям (да и организациям), взявшим на вооружение упрощенный, лишенный математической строгости подход к структурному программированию, не раз пришлось столкнуться с горьким разочарованием.

Если рассуждения Дейкстры об объеме доказательств корректности в структурном программировании представляются нам вдохновенным пророчеством, то тому есть две причины.

1. Доказательство корректности программы представляет собой единственно приемлемое описание для необходимой и достаточной документации этой программы. Здесь не требуются никакие дополнительные "идеи со стороны", и само доказательство является достаточным свидетельством того, что рассматриваемая программа удовлетворяет своей спецификации.
2. Объем доказательства корректности представляет собой хотя бы отчасти некоторый критерий сложности программы. Например, корректность длинной программы с небольшим числом переходов, возможно, будет проще доказать, чем корректность короткой программы с большим числом циклов, при этом длинная программа может оказаться к тому же проще. Или другой пример: с помощью хитроумных манипуляций с переменными и операциями можно уменьшить число переходов, однако доказательство после этого станет длиннее.

Как бы то ни было, понимание всего этого недоступно программисту до тех пор, пока он не уяснит, что же представляют собой доказательства корректности. К слову сказать, именно это обстоятельство и побудило меня взяться за статью "Как писать корректные программы, не сомневаясь в их корректности" [9]. Затем, вне зависимости от того, доказана ли корректность структурированных программ, это понимание безусловно поможет сократить их сложность и лучше их документировать.

Фактически рассуждения Дейкстры показывают, что идея математической корректности программ не выводится из концепции структурного программирования, а по времени появления даже предшествует ей (эта идея была намечена уже в работах фон Неймана и Тьюринга).

Но для большинства программистов того времени понятие математической корректности программ было неизвестным и казалось чем-то странным. Любопытно, что хотя появление первых компьютеров было вызвано и объяснялось необходимостью решения численных проблем математики (таких, как расчет баллистических таблиц), составление программ на этих компьютерах не рассматривалось большинством как занятие математическое по своей природе.

Ну а когда обнаружилось, что компьютеры можно использовать для обработки деловой информации, где дело сводится главным образом к текстовым (символьным) данным и

элементарной арифметике, связь между программированием и математикой стала казаться еще более слабой.

Как показал проект Skylab, писать структурированные программы можно не только на языках высокого уровня. Когда к структурному программированию подходили с точки зрения синтаксиса и не пытались глубже вникнуть в суть проблемы, могло показаться, что этот метод программирования тесно связан с языками высокого уровня. На деле это не так. Конечно, применение языков высокого уровня тоже привело к повышению производительности труда программистов, но это уже совсем иной вопрос. Итак, идеи структурного программирования, математической корректности и языки высокого уровня не зависят друг от друга.

Функции программ и корректность. Можно принять за правило, что математическая функция, которая переводит данные из некоторого начального состояния в конечное, реализуется в программе, завершающейся после своего выполнения (terminating program), — независимо от того, считается ли математической решаемая проблема.

К примеру, программа, создающая платежную ведомость, определяет математическую функцию точно так же, как и программа обращения матрицы. Даже программы, не завершающиеся после своего выполнения (non-terminating programs), такие, как операционные системы и системы связи, можно представить как единый незавершающийся цикл, бесконечно выполняющий завершающиеся подпрограммы.

Функция, описанная любой такой завершающейся программой, представляет собой не более чем набор упорядоченных пар — начальное и конечное состояния данных — которые могут возникнуть в процессе ее выполнения. Представление о том, что в операциях обращения матрицы "больше математики", чем в создании платежной ведомости, — это не более чем свойственная человеку культурная иллюзия; компьютерам она неизвестна, и они ее, соответственно, не разделяют.

Поскольку программы описывают математические функции, которые таким образом абстрагируются от всех деталей исполнения (включая даже такие детали, как используемый язык или компьютер), корректность программы относительно ее спецификаций можно рассматривать как чисто математическую проблему. Такая спецификация представляет собой отношение (relation). Если спецификация не допускает двусмысленного толкования корректного конечного состояния для заданного начального, то эта спецификация будет являться функцией.

Например, спецификация квадратного корня, требующая ответа, точного до восьмого десятичного разряда (так что все последующие разряды могут быть произвольными), — это отношение. Но спецификация упорядочения (sort specification) допускает только один конечный вариант расположения любого изначально заданного набора значений и, следовательно, является функцией. Программа будет считаться корректной относительно спецификации в том и только в том случае, если для каждого начального значения, допускаемого спецификацией, она создает конечное значение, соответствующее данному начальному значению в спецификации.

Поясним это на примере. Пусть функция f описывается программой P, а отношение r есть некая спецификация (r, возможно, является функцией). Тогда программа P корректна относительно отношения r в том и только в том случае, если справедливо некоторое уравнение корректности между f и r, например: domain(f $\cap$ r) = domain(r).

Чтобы убедиться в этом, обратите внимание, что f $\cap$ r состоит как раз из этих пар r, корректно вычисленных программой P, так что domain(f $\cap$ r) состоит из всех начальных значений, для которых P вычисляет корректные конечные значения. Но domain(r) представляет собой просто набор начальных значений, для которых r специфицирует приемлемые конечные значения, поэтому он должен быть равен domain(f $\cap$ r).

Такое уравнение в равной степени применимо и к программе, создающей платежные документы, и к программе обращения матрицы. Обе они могут быть математически корректными независимо от того, считают ли люди соответствующие расчеты математической процедурой.

Для иллюстрации этого уравнения корректности мы можем показать f и r на диаграмме Венна (Venn diagram) с проекциями этих множеств упорядоченных пар на множества их области

значений. Уравнение корректности требует, чтобы эти множества двух областей значений D(f ∩ r) и D(r) должны совпадать.

Доказательства математической корректности. В принципе, прямой путь доказательства математической корректности программы ясен. Начнем с программы P и ее спецификации r. На основе P определим ее функцию f и выясним, справедливо ли уравнение корректности между f и r.

На практике, если мы возьмем для примера "программу-спагетти", такое доказательство может оказаться неудобным — даже невозможным — из-за сложности программы. Но доказать, что корректна структурированная программа с той же функцией r, гораздо проще благодаря упорядоченности ее структуры управления. Ретроспективно причина этого кроется в алгебре функций, которая может ассоциироваться со структурным программированием.

В принципе легко увидеть, почему для функции программа является правилом. Для каждого начального состояния, из которого программа завершается нормально (не завершается аварийно и не инициирует бесконечных циклов), определяется уникальное конечное состояние.

Но в отличие от правил классических математических функций (устанавливаемых, например, полиномиальными выражениями, тригонометрическими выражениями и т.п.), определяемые программами правила для их функций могут быть совершенно произвольными и сложными. Хотя конечное состояние и является уникальным, возможно, что описать его будет непросто из-за сложных отношений зависимости между отдельными командами.

Единственный рациональный способ рассматривать "программу-спагетти" в качестве правила для функции состоит в том, чтобы представить, как она будет выполняться при некоторых конкретных данных, то есть смоделировав ситуацию в уме. Для небольших программ может подойти ограниченная родовая (generic) модель (например: "при отрицательных значениях программа выполняется в этом разделе").

Но в случае со структурированной программой существует более плодотворный способ ее представления: в виде функционального правила (function rule), построенного на использовании более простых функций. Например, любое следование, выбор или цикл определяет правило для функции, которая использует функции своих составных частей.

Алгебра частичных функций (algebra of part functions). Строить такие функции из вложенных частей структурированной программы интересно тем, что правила их конструирования очень просты и логичны. Они просто описываются как операции в определенной алгебре функций.

Правила для отдельных команд зависят от языка программирования. Например, правило для оператора присваивания x := y + z состоит в том, что конечное состояние совпадает с начальным с тем исключением, что значение, придаваемое идентификатору "x" изменяется на величину, придаваемую идентификатору "y" плюс значение, придаваемое идентификатору "z".

Правило для следования — это суперпозиция функций функций. Например, если операторы s1,s2 имеют функции f1,f2, функция последовательности s1;s2 будет суперпозицией f1'f2 = <x,y>: y = f2f1(x).

Важно отметить, что правила для каждого уровня используют *функции* на соседнем, более низком уровне, а не *правила* на соседнем более низком уровне. То есть, особый программный фрагмент определяет правило функции, но само правило не применяется на более высоких уровнях. Это означает, что любой фрагмент программы может быть надежно заменен произвольным образом на другой, имеющую ту же функцию, даже если последняя представляет совсем иное правило.

Например, программные фрагменты x := y и IF x <> y THEN x := y определяют разные правила для одной и той же функции и могут произвольно заменяться один на другой.

Аксиоматическая и функциональная верификация. Сравнивая подход к этим проблемам, сформировавшийся в университетах и в компьютерной индустрии, с удивлением обнаруживаешь, что на сегодняшний день сложилась довольно парадоксальная ситуация. Хотя доказательства

корректности программ изучаются во многих университетах применительно к игрушечным программам, большинство преподавателей, не занимающихся предметом вплотную, считают проблему корректности программ чисто академической. Студентам этот материал дают из соображений общей культуры: "На практике этим вам никогда не придется заниматься, но знать, как это делается, все же полезно".

С другой стороны, именно вокруг идеи корректности программ строятся учебные планы Института программной инженерии IBM (IBM Software Engineering Institute) и как раз потому, что эта идея не является чисто академической. Напротив, она предоставляет в распоряжение специалиста практический метод рассуждения о больших программах, который ведет к значительному улучшению качества и повышению производительности при разработке программного обеспечения.

Впрочем, парадокс объясняется просто. Университетские профессора знакомят студентов в первую очередь с такой формой корректности программ, которая называется аксиоматической верификацией и которая может непосредственно применяться к игрушечным учебным программам. А в Институте программной инженерии IBM изучается другая форма, именуемая функциональной верификацией, и материал подается таким образом, чтобы его можно было применять к большим программам.

В случае с аксиоматической верификацией корректность программы доказывается с помощью рассуждений о воздействии программ на данные. Обоснование при этом принимает форму предикатов над данных (predicates on data), содержащихся в различных частях программы и остающихся инвариантными в ходе ее выполнения. Отношения между этими предикатами заданы аксиомами языка программирования (отсюда и название), и предикаты входа/выхода (exit/entry predicates) совместно описывают функцию программы в альтернативной форме. Тони Хоар (Tony Hoare) дал изящное объяснение этих рассуждений, представив их как форму естественной дедукции, именуемую ныне логикой Хоара (Hoare logic) [11].

Функциональная же верификация с самого начала основывается на теории функций. Например, простой оператор присваивания x := y + z описывает функцию, которую можно обозначить как [x := y + z], а затем использовать как функцию в алгебре частичных функций структурированных программ (algebra of structured-program part functions). На практике обучать функциональной верификации сложнее, но ее легче применять к большим программам, поскольку алгебраическая структура выражена в ней в явной форме.

Самое важное различие между аксиоматической и функциональной верификацией проявляется в при работе с циклами. Когда корректность программы проверяется с помощью аксиоматической верификации, для каждого цикла должен быть задан инвариант цикла. А когда используется метод функциональной верификации, такие инварианты в ходе пошагового уточнения не нужны, потому что они уже воплощены для каждого цикла в функции или отношении его спецификации [8].

Аксиоматическую верификацию можно легко объяснить непосредственно, с точки зрения переменных программы и воздействия на них операторов применительно к конкретным особенностям любого данного языка программирования. Но когда программы становятся большими, растет и число их переменных — а число функций при этом не изменяется. Так что теория, не привязанная к числу переменных, легко применяется при усложнении функции, но не при значительном увеличении числа переменных.

Такую функцию можно уместить в двух строчках математических символов; ее можно описать и с помощью обычного английского текста (но уже на сотне страниц) — в любом случае математическая форма этой функции будет представлять собой множество упорядоченных пар. Понятно, что сто страниц текста содержит гораздо больше двусмысленностей и погрешностей, но если хорошо организовать работу бригады проектировщиков, возросший риск совершения ошибки тем или иным программистом можно снизить с помощью системы проверок и противовесов. Так что вообще отказываться от этой методологии нет никаких оснований.

В результате, функциональная верификация состоящего из 100 000 строк проекта верхнего уровня и верификация проекта нижнего уровня, состоящего из 10 строк, идентичны по своей форме. Существует одно функциональное правило, подлежащее верификации с помощью небольшого числа функций на следующем уровне. Эта функция описывает преобразование из начальных состояний в конечные. Эти состояния в конечном счете будут представлены как

наборы значений переменных, но их можно рассматривать как абстрактные объекты непосредственно в проектировании высокого уровня.

Хотя эти рассуждения ведутся по большей части на естественном языке соответствующей прикладной программы, их правила определяются алгеброй функций, которая хорошо описана с математической точки зрения и понятна большинству как разработчиков, так и лиц, осуществляющих проверку.

Есть серьезные основания полагать, что такие нестрогие рассуждения в математической форме могут быть эффективными и надежными при работе с объемными программными системами (свыше миллиона строк), которые проектируются и разрабатываются с помощью нисходящего подхода, причем бэктрекинг (перебор вариантов с возвратами) при проектировании приходится применять крайне редко [12].

Имеется еще один способ описания рассуждений, необходимых для доказательства корректности структурированных программ. Предикаты переменных программы, связанных с аксиоматической верификацией допускают применение алгебры предикатов, операций которой в классической книге Эдсгера Дейкстры были названы преобразователями предикатов (predicate transformers) и позднее описаны более подробно в замечательной работе Дэвида Гриса (David Gries) [13].

Взгляд в будущее

Программирование со структурированными данными. Задача сокращения объема формального доказательства корректности может быть поставлена применительно к структурированным программам, и это дает удивительные и конструктивные результаты. При доказательстве корректности структурированных программ алгебраические операции с соответствующими функциями идентичны для каждого уровня, но в верхних частях иерархии эти функции становятся более сложными.

Объем формального доказательства в значительной степени зависит от двух особенностей данных анализируемой программы: (1) от числа переменных программы, описывающих данные; и (2) от присваиваний в массивах.

Массивы воплощают собой произвольный доступ к данным точно так же, как операторы "goto" олицетворяют произвольный доступ к командам. Цена, которую приходится платить за такой доступ, выражается непосредственно в объеме и сложности доказательств, связанных с изменением значений элементов массива.

Изменение значения элементов массива, например, x(i)y:= y(j+k) соотносится с тремя предыдущими присваиваниями значений элементам i, j и k. Значения i или j + k могут выходить за пределы области значений функции, а если они остаются внутри диапазона, это обязательно нужно объяснить. Более того, массив "x" будет изменен в i-ом элементе, и этот факт должен быть объяснен в следующий раз, когда будет вновь осуществлен доступ к "x" для выяснения того же i-го значения (которое может быть значением другой переменной m).

Отношение к массивам самого Дейкстры [14] представляет собой очень яркое свидетельство их сложности. Кстати говоря, для изменения значений элементов массива Грис тоже применял преобразователи предикатов [13], гораздо более сложные, нежели те, что использовались им для простых присваиваний.

К счастью, существует возможность "одним махом" решить проблему этих расширителей доказательств (proof expanders): исключить использование массивов в структурном программировании и применять вместо них абстракции данных без произвольного доступа. На ум тут же приходят три подобные простые абстракции: множества (set), стеки (stack) и очереди (queue), причем две последние структуры данных — с алгоритмами доступа в обратном (LIFO) и прямом (FIFO) порядке поступления. Для присваивания значений элементам стеков или очередей (или другим элементам — значений данных из стеков или очередей) указатели не требуются, поэтому в таких присваиваниях можно обойтись меньшим количеством переменных.

Более того, доказательства, связанные с изменением значений элементов множеств, стеков и очередей, гораздо короче доказательств, основанных на применении массивов. Конечно, для того чтобы разработать программу без использования массивов, надо приложить намного больше усилий — точно так же, как гораздо больших усилий требует программирование без применения оператора "goto". Но в этом случае из-под пера разработчика выходят более продуманные программы, корректность которых доказывается проще. А на одну команду в такой программе приходится больше функций, нежели в программе, построенной с использованием массивов.

Так, присваивание "массив — массив" x(i) := y(j + k) представляет собой всего лишь одну из четырех команд, необходимых для пересылки элемента данных из "y" в "x" (присваивания требуются также для x, i, j и k).

С другой стороны, присваивание "стек — очередь", например, back(x) := top(y) пересылает вершину стека "y" в хвост очереди "x" без предварительных присваиваний. Разумеется, необходимо более тщательное планирование, чтобы в вершине стека "y" оказался тот самый элемент, который нужно переслать в хвост очереди "x".

Этот алгоритм доступа к данным, в котором место массивов заняли стеки и очереди, использовался в процессе проектирования сложной системы линвистической обработки, содержащей порядка 35 000 строк [2]. Независимые оценки ее объема показывают, что на каждую команду здесь приходится почти в пять раз больше функций, чем можно было бы ожидать от программы, построенной на использовании массивов.

Данный проект был полностью верифицирован (важно отметить, что тестирование системы проводилось без какой-либо предварительной отладки). В ходе проверки было установлено, что на тысячу команд в программе приходится 2,5 ошибки, причем все они были легко обнаружены и исправлены. После тестирования ядро системы (порядка 20 000 команд) работало в течение двух лет без выявления каких-либо ошибок.

Функциональная верификация вместо отладки компонентов. Функциональная верификация структурированных программ позволяет создавать высококачественное программное обеспечение без отладки его компонентов. Отладка компонентов программы кажется нам необходимой точно так же, как кажется необходимым применение операторов "goto" и массивов. Однако опыт практического применения функциональной верификации показал, что разработчики могут создавать программное обеспечение без использования отладки и с весьма хорошими результатами.

Эта дремлющая в программистах способность использовать функциональную верификацию дает удивительный эффект в сочетании со статистическим тестированием на системном уровне — то есть с таким тестированием, когда на ввод подаются статистически сгенерированные данные, подобные тем, которые позднее будут поступать в систему от ее пользователей. Статистическое тестирование как метод разработки применяется нечасто, и тому есть серьезные причины. Ведь речь идет о программном обеспечении, которое требует значительной работы по устранению недостатков, которая направлена лишь на то, чтобы заставить его просто работать, не говоря уже о том, чтобы заставить его работать надежно. И все же статистическое тестирование функционально верифицированных и структурированных программ и в самом деле дает высокие результаты.

Проектирование методом стерильного цеха (cleanroom). Комбинированный алгоритм, при котором программа подвергается статистическому тестированию без предварительной покомпонентной отладки, называется проектированием программного обеспечения методом стерильного цеха. Термин "стерильный цех" отражает упор не на устранение, а на профилактику дефектов (так же организовано и производство аппаратного обеспечения, но здесь метод примененяется в процессе проектирования, а не при производстве продукта).

Фактически проектирование программного обеспечения методом стерильного цеха позволяет разрабатывать программное обеспечение при статистическом контроле качества с помощью итеративной наращиваемой разработки и тестирования. Результаты, полученные в ходе первых шагов, могут быть подвергнуты статистическому тестированию с целью научной оценки их качества и обеспечения возможности корректировать процесс разработки для получения в дальнейшем результатов необходимого качества.

На первый взгляд, отказ от покомпонентной отладки представляется странным: ведь кажется, что такая отладка — это простой и естественный способ устранить большую часть дефектов, которые могут содержаться в программном обеспечении. Но отладка — процедура весьма коварная. Следуя ее логике, разработчик сосредоточивается на одной проблеме, и вынужден выпускать из виду все остальные. Поэтому в ходе покомпонентной отладки возникает опасность получить серьезную системную ошибку после исправления простого просчета. А ведь сама идея покомпонентного тестирования (unit testing) предполагает отладку, которая, в свою очередь, снижает тот уровень концентрации внимания и дисциплины проектировщиков, на который можно расчитывать при иных обстоятельствах.

Мало того, отказ от покомпонентного тестирования и отладки дает несколько преимуществ:

- более серьезное внимание к проектированию и верификации, которые воспринимаются каждым программистом как неотъемлемые аспекты его личной работы;
- более серьезное внимание к проверке проектирования и верификации со стороны команд разработчиков;
- сохранение проектной гипотезы для статистического тестирования и контроля (отладка компрометирует проект);
- отбор квалифицированного персонала, способного создавать удовлетворительные программы без отладки;
- высокая мотивация квалифицированного персонала.

С другой стороны, проведенное с позиций пользователя статистическое тестирование не подвергавшегося отладке программного обеспечения дает несколько преимуществ:

- научно обоснованные оценки надежности программного обеспечения и дополнительные ресурсы надежности после обнаружения и исправления ошибок в ходе системного тестирования;
- обязательное для всех программистов условие в полном объеме соблюдать спецификации пространства ввода и проекта программы на основе разбиения спецификаций (такие ситуации, когда разработчики сначала доводят систему до рабочего состояния и лишь потом берутся за логику обработки исключений, при этом не возникают);
- наиболее эффективный метод повышения надежности программного обеспечения с помощью тестирования и исправления ошибок.

Есть все основания полагать, что разработчики систем на промышленных предприятиях в состоянии создавать программное обеспечение беспрецедентно высокого качества. Вместо того чтобы писать программы, где на тысячу строк кода приходится 50 ошибок и затем в процессе отладки сокращать число их на 90%, оставляя тем самым по пять ошибок на тысячу строк, программисты, пользующиеся методом функциональной верификации, могут создавать код, который никогда не выполнялся с числом ошибок более 5 на тысячу строк и устранять почти все эти ошибки в ходе статистического тестирования системы.

Более того, существует качественное различие между ошибками, обнаруживаемыми после функциональной верификации, и ошибками, остающимися после отладки. Ошибки первой категории объясняются математическими погрешностями и представляют собой банальные просчеты при кодировании — просчеты, которые легко обнаруживаются в ходе статистических тестов.

Границы человеческих возможностей. Эксперты не устают изумляться заложенной в людях способности усваивать новые технологии. Вспомним, что 70 лет назад специалисты с уверенностью предсказывали наступление того дня, когда скорость серийных автомобилей достигнет 70 миль в час. Но кто из них мог предположить, что управлять этими машинами будут семидесятилетние бабушки?!

Тридцать лет назад эксперты предсказывали, что компьютеры завоюют мировую шахматную корону, но в то же время они мало что предсказывали относительно программистов, которым предстояло решить эту задачу (в сущности, предсказание было одно: путь к созданию программ, способных превратить компьютеры в шахматных чемпионов, — это путь новых проб и ошибок). И в этом проявилась та легкость, с какой мы обычно переоцениваем потенциал машин и недооцениваем возможности людей. Компьютеры пока так и не стали шахматными королями,

тогда как успехи программистов в плане строгости логических решений превосходят все ожидания.

С первых дней компьютерного программирования стало аксиомой считать, что ошибки являются необходимой частью любой программы, поскольку человеку свойственно ошибаться. Это утверждение неоспоримо, но если не вводить количественных характеристик, из него вряд ли можно извлечь много пользы. И хотя сейчас модно оперировать числом ошибок на тысячу строк кода, лучше применять другой показатель: количество ошибок на человека в течение года работы по созданию программного обеспечения.

Такой показатель позволяет учитывать различие в сложности проектов — в программах с высокой степенью сложности на тысячу строк кода приходится больше ошибок, но и на написание тысячи строк кода в этом случае уходит больше времени. Предложенный здесь показатель выравнивает различие в сложности программ и дает еще то преимущество, что ошибки соотносятся со временем, а не с результатом, что является более фундаментальным подходом.

Так, при разработке системы для New York Times каждый прогрпаммист совершил в среднем одну ошибку за год работы. До того времени такое качество считалось недостижимым, но сегодня этот показатель постоянно улучшается усилиями квалифицированных программистов.

Еще более высоких результатов добился Пол Фрайдей (Paul Friday), разработавший в 1980 г. систему программного обеспечения для распределенного управления сетью связи и сбора данных по национальной переписи населения. Эта система реального времени состояла из порядка 25 000 строк, была разработана методом структурного программирования с использованием функциональной верификации и работала на протяжении всей переписи (почти год) без единой ошибки. За свое достижение Фрайдей был удостоен золотой медали, высшей награды Департамента торговли (в ведении которого находится Бюро переписи). Специалисты по промышленному программному обеспечению, изучившие реализованные в этом продукте функции, пришли к выводу, что решение проблемы в границах 25 000 строк весьма экономично (как представляется, высококачественное, функционально верифицированное программное обеспечение предоставляет больше функций на строку кода, чем созданное на основе других методов проектирования).

Если исходить из того, что при разработке программного обеспечения средней сложности программист в среднем пишет 2 500 строк кода в год и на 10 человеко-лет труда приходится одна ошибка, то система из 25 000 строк кода должна содержать одну ошибку. С другой стороны, такая система с достаточной степенью вероятности может оказаться вообще безошибочной. Этот уровень уже достигнут. Можно ожидать, что использование структурного программирования, функциональной верификации, а также проектирования по методу стерильного цеха (и хорошая организация работы, конечно) позволят в предстоящее десятилетие повысить этот параметр работы программистов еще на один порядок.

Структурный подход освободил программирование от многих ненужных сложностей и способен еще открыть перед нами новые широкие горизонты. Но все же сделать предстоит многое. Ведь мало научить студентов функционально верицицировать корректность игрушечных учебных программ — они должны освоить этот метод применительно к большим и реальным программам. Недавно в свет вышло учебное пособие для студентов [15], в котором предпринята попытка решить эту проблему.

Людям свойственно ошибаться, и чтобы защитить программирование от этого недостатка человеческой натуры, требуются более совершенные средства разработки программного обеспечения. Но вот чего явно не нужно, так это интерактивного отладчика. Ведь вместо того чтобы содействовать развитию систематического проектирования, такой инструмент поощряет любительские изыски по принципу "не получится так — попробуем эдак" и обеспечивает спокойную жизнь людям случайным, плохо подготовленным для такого занятия, как тщательное программирование (precision programming). Гораздо полезнее подумать над разработкой организатора доказательств (proof organizer) и соответствующего тестера (checker). Администраторам в промышленности, ответственным за контроль производительности труда программистов, не обойтись подсчетом строк кода — толку от этого не больше, чем от подсчета слов, произнесенных за день работы продавцом в магазине. Чтобы рационально оценивать работу программистов, администраторы должны лучше понимать существо проблемы. А

повышение производительности и улучшение качества требует уже дополнительных инвестиций в образование и в разработку инструментальных средств.

Но главная проблема администраторов состоит в том, чтобы объединить хорошо образованных инженеров-програмистов в эффективные команды разработчиков. Люди не могут не совершать ошибок — это верно, но из этого часто делают слишком мрачные выводы, которые совсем не оправдывают себя в хорошо организованных коллективах, опирающихся в своей работе на систему проверок и противовесов.

Литература

[1] Edsger W. Dijkstra (1969) Structured Programming // Software Engineering Techniques // NATO Science Committee, Rome, p.88-93.

[2] Harlan D. Mills, Richard C. Linger (1986) Data Structured Programming: Program Design without Arrays and Pointers // IEEE Transactions on Software Engineering, Vol.12, No.2, p.192-197.

[3] Paul A. Currit, Michael Dyer, Harlan D. Mills (1986) Certifying the Reliability of Software // IEEE Transactions on Software Engineering, Vol.12, No.1, p.3-11.

[4] O.J.Dahl, Edsger W. Dijkstra, C.A.R.Hoare (1972) Structured Programming // Academic Press.

[5] Corrado Boehm, Giuseppe Jacopini (1966) "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules // Communications of the ACM, Vol.9, No.5, p.366-371.

[6] F.Terry Baker (1972) Chief-Programmer Team Management of Production Programming // IBM Systems Journal, Vol.1, No.1, p.56-73.

[7] F.Terry Baker (1972) System Quality Through Structured Programming // AFIPS Conference Proceedings, FJCC, Part 1, p.339-343.

[8] Richard C. Linger, Harlan D. Mills, Bernard I. Witt (1979) Structured Programming: Theory and Practice // Addison-Wesley.

[9] Harlan D. Mills (1983) Software Productivity // Little, Brown, and Co., Boston.

[10] Niklaus Wirth (1971) Program Development by Stepwise Refinement // Communications of the ACM, Vol.14, No.4, p.221-227.

[11] C.A.R.Hoare (1969) An Axiomatic Basis for Computer Programming // Communications of the ACM, Vol.12, No.10, p.576-583.

[12] Anthony J. Jordano (1984) DSM Software Architecture and Development // IBM Technical Directions, Vol.10, No.3, p.17-28.

[13] David Gries (1981) The Science of Programming // Springer-Verlag.

[14] Edsger W. Dijkstra (1976) A Discipline of Programming // Prentice-Hall.

[15] Harlan D. Mills et al. (1987) Principles of Computer Programming: A Mathematical Approach // Allyn and Bacon, Rockleigh, New York.

Об авторе. Харлан Миллз (Harlan D. Mills) — на момент написания статьи сотрудник отделения федеральных систем корпорации IBM (IBM Federal Systems Division) и профессор компьютерных наук в университете Мэриленд (University of Maryland, USA). До поступления в штат сотрудников IBM в 1964 г. Миллз работал в General Electric и в RCA, был президентом компании Mathematica. В 1975 г. приступил к работе в университете Мэриленд, перед этим поработав в университете шт.Айова (Iowa State University), Принстонском университете (Princeton University) и в университете Джонса Хопкинса (Johns Hopkins University). Диссертацию в области математики защитил в университете шт.Айова. Возглавлял Компьютерное общество IEEE (IEEE Computer Society) и был руководителем Фонда DPMA (DPMA Education Foundation). В 1985 году награжден научной премией DPMA's Distinguished Information Science Award.