9.6.82

Кибернетика № 6, 1982

Два облика программирования

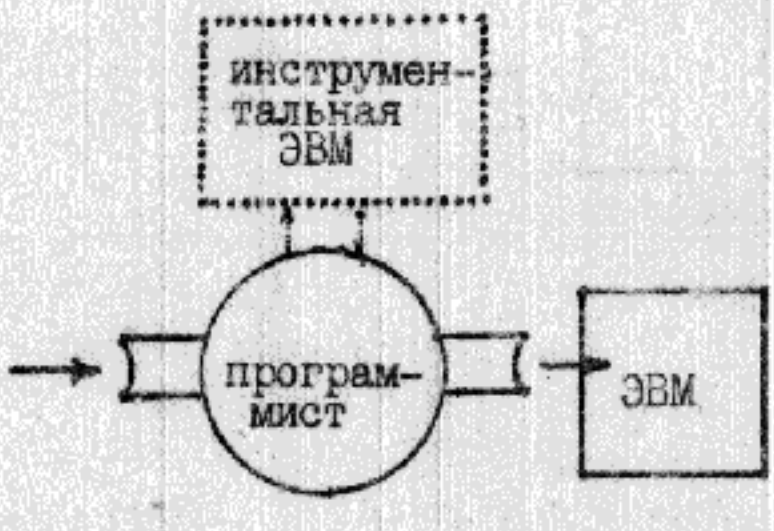
295-39

А.П.Ершов

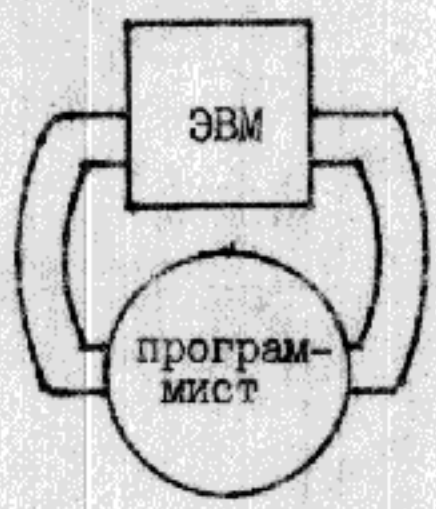
Вычислительный центр СО АН СССР
Новосибирск 630090, СССР

Эта заметка появилась на свет при размышлении о разработке программного обеспечения как человеческой деятельности. Если в этом плане рассматривать работу программиста, то нужно различать два, как мне кажется, весьма разных ее вида: к одной из них программист относится как слуга, а к другой - как хозяин.

Разовьем этот тезис подробнее.



Программист-слуга



Программист-хозяин

Когда я думаю о программисте как о хозяине, я имею в виду, что он программирует для себя. Имея все ресурсы, все средства (виртуально или физически - неважно!) в своем распоряжении, он является единственным и окончательным судьей своим действиям и их результату.

Когда я думаю о программисте-слуге, то он мне представляется прежде всего в виде канала связи, воспринимающего предъявленную ему спецификацию задачи. Ответственность программиста за правильность спецификации весьма ограничена, с другой стороны он принимает на себя обязательство старательно реализовать принятые спецификации и выдать клиенту программу-продукт (для разового счета или постоянного применения - неважно!).

295-40

Естественно, что это различие замечалось многими. Ф.Л.Бауэр [1] называет работу программиста-слуги программированием по контракту. Соответственно можно назвать работу программиста-хозяина программированием для себя. Э.Сандевал [2] развивает близкий подход, выделяя группу "оконечных" программистов. Иногда это различие проводят, употребляя для слуги и хозяина термины профессионального и непрофессионального программирования соответственно. Такая трактовка допустима, если мы будем исследовать социальную сторону программирования как деятельности, например, его профессиональную этику. Если же говорить о программировании, имея в виду его внутреннее содержание, то в этом случае взгляд на программиста-хозяина как на непрофессионала может привести к недоразумениям.

Программист-слуга, в некотором (очень глубоком) смысле, не знает, чего хочет заказчик, и полагается исключительно на формулировку проблемы. Конечно, он не безграмотен, и может знать кучу вещей о предметной области. Он может оказать заказчику неоценимые услуги, обнаружив много несообразностей в его спецификациях. Однако, все это сверх плана. Его главное и единственное дело - преобразовать спецификацию в надежный и эффективный результат: программу или данные.

Это тот облик программирования, над которым трудились больше всего и который доминировал в период становления программирования. Этот труд привел к возможности строго обосновывать каждый шаг построения программы, и эта возможность, без сомнения, является крупнейшим достижением вычислительной науки. Методологический принцип абстрагирования от содержательного знания программируемой задачи оказался очень плодотворен: он привел к разработке общезначимых и мощных методов манипулирования программами и рассуждения о них, опираясь на схемное представление программ в неинтерпретируемой сигнатуре элементарных операций и предикатов.

Другой облик программирования, тем не менее, тоже существовал все это время. Будучи повернутым в сторону задворок программирования, он однако набирает "второе дыхание" в связи с появлением и развитием персональных ЭВМ. Мало того, системный программист на своей кухне, уставленной инструментальными машинами, часто ведет себя как хозяин. В этом хозяйстве уже накопились

295-41

такие средства, как расширяемые языки, макропроцессоры, переписывающие системы, операции периода компиляции, универсальные редакторы и многое другое. Этот ассортимент, однако, сложился случайно и не упорядочен ни хорошей теорией, ни надежной методологией. Только недавно появился собирательный термин, под который, как под знамя, собираются бессапожные ситтемные программисты, - программная обстановка (programming environment, или - более развернуто, но зато более точно для русского языка - операционная обстановка для построения программ. Превратить это модное словоупотребление в стройную методологию с солидным теоретическим обоснованием - важная и увлекательная задача теоретического и системного программирования 80-х годов.

Программист-хозяин знает, что ему надо. В этом - коренная методологическая разница между программированием по контракту и программированием для себя. Этот принцип эмпирически нащупали разработчики экспериментальных персональных ЭВМ, которые без колебаний отдали приоритет действию перед планом. Как выразился Дж.Аттарди [3]: "видеть и действовать, а не запоминать и писать". Естественно, что адепты строгого подхода воспринимают это как ересь, однако на самом деле - это новое явление в программировании, которое требует теоретического осмысления и проработки.

Цель этой заметки - изложить вариант теоретической модели программной обстановки, в которой работает программист-хозяин. Эту модель я называю трансформационной машиной (ТМ). Машина поддерживает различные преобразования (трансформации) пар "программа-данные" в пары "программа-данные". Правильно устроенная ТМ сохраняет функциональный инвариант преобразуемых пар, т.е. если $(p',d')=t(p,d)$, где t - трансформация, выполненная машиной, то $p'(d')=p(d)$.

Другим важным свойством трансформационной машины является то, что она строится самим программистом. Ее построение является иерархической конструкцией, и выделенный уровень этой иерархии я, следуя Э.Сандевалу, А.А.Берсу и П.К.Леонову, буду называть контекстом. Контексту доступны преобразуемая пара (p,d) и набор базовых преобразований, каждое из которых осуществляет элементарное (на этом уровне) преобразование пары (p,d) в другую пару (p',d') с сохранением функционального инварианта. Базовые

преобразования делятся на три категории - редукции, раскрытия и конверсии. Редукции имеют дело с интерпретацией элементарных (на этом уровне) операций и предикатов, раскрытия раскрывают составные конструкции, а конверсии носят схемно-комбинаторный характер. Примерами редукции являются, например, замена конструкции если ист то A иначе B все на A или 3+5 на 8. Пример раскрытия - реализация вызова процедуры путем ее открытой подстановки, а пример конверсии - перераспределение памяти или экономия совпадающих подвыражений.

Что важно? Важно, что базовые трансформации могут быть применены свободно. Их применение в произвольной последовательности не нарушает функционального инварианта преобразуемой пары. Это разрешает главную трудность: программист может сделать то, что хочет, не боясь ошибиться. Этого мало. Редукции с раскрытиями и конверсии обладают определенным свойством полноты. Полнота редукций и раскрытий позволяет находить единственным образом нормальные формы пар (p,d), получая их как неподвижные точки базовых трансформаций. Если функциональным инвариантом пары (p,d) является константа $c = p(d)$, то тогда нормальной формой этой пары будет пара (ϕ, c) с полностью редуцированной программой и с вычисленным результатом в поле данных. Если инвариантом пары (p,d) является некоторая функция $\varphi(y)$, то тогда нормальной формой для (p,d) будет пара, программная компонента которой содержит некоторую остаточную программу для φ, а компонента данных содержит имя аргумента y и (может быть) некоторые константы и имена промежуточных величин.

Конверсии тоже могут обладать свойством полноты по отношению к некоторому схемному инварианту или канонической форме программы и ее данных.

Иерархия контекстов достигается тем, что нахождение неподвижных точек редукций или канонических форм конверсий трактуется как атомарный акт в контексте следующего уровня, а базовые трансформации контекста раскрываются средствами внутренних контекстов. Добавим также, что трансформационная машина представляется удобной концепцией для реализации абстрактных типов данных, в которых описания типов и операций реализуются раскрытиями, а редукции и конверсии соответствуют аксиомам.

То обстоятельство, что иерархия контекстов возникает и строится по воле программиста, позволяет ему осуществлять очень точную настройку программного процессора на соотношение между компиляцией, интерпретацией и прямым вычислением, делая эти разграничения подвижными, гибкими и управляемыми.

Конечно, правила преобразований, придумываемые программистами, не могут быть произвольными. Монитор программной обстановки должен воплощать и поддерживать некоторое знание о функциональной зависимости, общих правилах композиционной иерархии, именовании, управляющих и информационных связях. Это общее знание позволит на любом уровне поддерживать абстрактные инварианты, гарантирующие сохранение функциональных инвариантов.

Изложенные в этой заметке соображения развивают исходный тезис в русле трансформационного подхода к программированию. Его особенность состоит в том, что направление манипуляций с программами не столько требует предпланирования, сколько определяется складывающейся обстановкой. Программист-хозяин, однако, только тогда сможет хорошо распорядиться предоставленной ему свободой действий, если будет хорошо видеть поле для их применения. Большой и четкий экран дисплея нужен программисту так же, как широкое и чистое ветровое стекло автомобилисту. Но если за ветровым стеклом действительность сама подставляет водителю дорожные знаки и ситуации, то для системного программиста нужно еще много потрудиться, чтобы превратить подслеповатые литеры алфавитно-цифровых дисплеев в компактное и наглядное изображение програм и данных. Хотелось бы обратить внимание читателя на некоторые новые принципы взаимодействия человека с машиной, выдвигаемые так называемыми объектно-ориентированными языками, из которых в первую очередь надо выделить язык Смолток [5].

Возвращаясь еще раз к различению двух обликов программирования, следует признать, что, опираясь в конце-концов на одно и то же устройство - ЭВМ, обе формы программирования математически вполне переводимы друг в друга. Тем не менее, как практическое, так и теоретическое их различие (чни) будет помогать делу, позволив снабдить обе категории программистов адекватными операционными средствами, методологическими установками и математическими теориями.

1. Bauer F.L. Programming as fulfilment of a contract.- In: "Infotech state of the art reports, Series 9, No.6. System Design".- Maidenhead: Pergamon Infotech, 1981, p. 167-174.

2. E.Sandewall. An environment for development and use of executable application models. - In: Records of the 2nd Software Technology Seminar, Capri, May 3-7, 1982, p. 12+43 p.

3. Attardi G. Office information systems desing and implementation. - (Technical report) Cnet No.47. Instituto di Scienze dell'Informazione Univers. di Pisa, Pisa, 1980, 44p.+ii.

4. Ершов А.П. Смешанные вычисления: потенциальные применения и проблемы исследования. - В кн.: Методы математической логики в проблемах искусственного интеллекта и систематическое программирование. Тезисы докладов и сообщений. Часть 2. В надзаг.: Ин-т математики и кибернетики АН ЛитССР. Вильнюс, 1980, с.26-55.

5. Ingalls D.H. The Smalltalk-76 programming system: design and implementation. - In: Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, 1978.