17

Eiffel

Eiffel – это объектно-ориентированный язык, разработанный в основном Бертраном Мейером (Bertrand Meyer) в 1985 году. Развитием этого языка сейчас управляет комиссия по стандартизации в Ecma International, выпустившая стандарт ISO в 2006 году. Он включает в себя набор современных и широко распространенных функций: сборка мусора, обобщенное программирование и безопасность типов. Вероятно, главным вкладом Eiffel стала идея «контрактного программирования» (design by contract), в соответствии с которой язык обеспечивает предусловия, постусловия и инварианты интерфейсов, что способствует надежности и повторному использованию компонентов. Влияние Eiffel очевидно в таких языках, как Java, Ruby и C#.

День вдохновения

Почему вы решили создать язык программирования?

Бертран Мейер: Очень редко языки программирования создаются из чистого интереса. Eiffel появился в силу необходимости. Я разработал язык программирования, потому что мне нужно было писать программы, а все другие языки меня не удовлетворяли.

Вам нужен был инструмент для реализации контрактного программирования?

Бертран: Это очевидная причина, но в целом мне нужен был объектно-ориентированный язык. Позвольте описать вам обстановку. В 1985 году мы открыли компанию Interactive Software Engineering. Сейчас она называется Eiffel Software. Мы собирались делать инструменты для программной инженерии. Одна японская компания выделила нам средства для создания редактора программ или редактора, управляемого синтаксисом, который был нами создан и имел некоторый успех.

Это была очень маленькая компания. Я все еще преподавал в Калифорнийском университете Санта-Барбары, так что это был несколько побочный бизнес. У нас были рабочие станции UNIX, полученные от японских заказчиков, для которых они были одним из видов продукции. Шел 1985 год, и я программировал в объектно-ориентированном стиле уже почти десять лет. В 1970-х годах мне повезло натолкнуться на язык Симула-67, которым я сразу увлекся. Я был уверен, что программировать нужно именно так.

На наших машинах не было компилятора Симулы, а я очень любил этот язык. Как сказал Тони Хоар об Алголе, он стал усовершенствованием многих своих наследников. Тем не менее в Симуле не было ни множественного наследования, ни универсальности – а к тому времени я понял, что нужно и то, и другое. Я обосновал это в докладе «Genericity versus Inheritance» (Универсальность и наследование), подготовленном для первой конференции OOPSLA. Мы решили посмотреть, что было доступно в то время. Имелся C++; я открыл книгу и быстро закрыл ее, потому что мне нужно было нечто иное – идея сделать объектность привлекательной для программистов Си была интересна, но явно могла быть только временным решением на пути к чему-то более цельному. Был также Objective-C, но он был слишком тесно связан со Smalltalk и плохо поддерживал те принципы программной инженерии, которые нас интересовали. То же самое относится к самому Smalltalk. Smalltalk – отличная разработка, но она никак не согласовывалась с нашими стремлениями. Eiffel появился как комбинация объектно-ориентированной технологии с принципами и практикой программной инженерии, разработанными в предшествующее десятилетие.

Smalltalk обладал отчетливыми чертами экспериментального программирования, что представлялось неуместным для тех задач, которыми мы собирались заниматься; например, отсутствие статических типов уже никуда не годилось для нас. Там была масса замечательных идей, но ничего из того, что нам было нужно.

Тогда я написал доклад. Это был доклад UC–Santa Barbara, который фактически описывал не язык, а библиотеку структур данных и алгоритм, потому что я был сильно увлечен повторным использованием и хотел, чтобы стандартная библиотека охватывала основные структуры компьютерной науки, которые я иногда называю «Knuthware». Позднее ее назвали EiffelBase, а тогда – просто библиотекой структур данных (Data Structures Library). В моем докладе описывались массивы, связанные списки, стеки, очереди и так далее. Там была использована особая нотация, и я заявил, что мы собираемся это реализовать. Я считал, что на реализацию уйдет три недели. Мы занимаемся ею до сих пор. Но это уже был Eiffel.

Язык не был самоцелью. Целью были многократно используемые компоненты, и я понимал, что для хороших многократно используемых компонентов нужны классы, нужна обобщенность, которую мы включили туда с самого начала, нужно множественное наследование и нужна взвешенная комбинация обобщенности и множественного наследования, что продемонстрировал мой доклад для OOPSLA. Нужны абстрактные классы. Нужны, конечно, контракты, что для меня было самым естественным делом. Все так переживают из-за них, а мне до сих пор непонятно, как можно программировать без контрактов. Еще я знал, что нужен хороший механизм потоков или сериализации, и работу над ним мы не откладывали на конец. Я понял это благодаря языку SAIL (Stanford Artificial Intelligence Language), который был очень хорошо спроектирован – не объектно-ориентированный, но очень интересный, – и с которым я работал в Стэнфорде десятью годами ранее. Принципиально важно было все это иметь с самого начала. Очевидно, конечно, что нужен был механизм сборки мусора.

Как вы пришли к этой философии? Ваш опыт практического программирования позволял определить, как можно усовершенствовать создание ПО?

Бертран: Отчасти опыт, отчасти, разумеется, чтение литературы. Будучи студентом Стэнфорда, я прочел в 1973 году книгу Даля, Дейкстры и Хоара «Structured Programming»[1]. На самом деле, это три монографии под одной обложкой. Первая, написанная Дейкстрой, – это знаме-

[1] Даль У., Дейкстра Э., Хоор К. «Структурное программирование». Серия: Математическое обеспечение ЭВМ. – Пер. с англ. – М.: Мир, 1975.

нитая книга о структурном программировании. Вторая, написанная Хоаром, рассказывает о структурах данных и тоже замечательна, но там была еще третья часть. Один из важных уроков, усвоенных мною из жизни, состоит в том, что люди читают начало книги, поэтому – фактически это совет тем, кто пишет книги, – очень тщательно выбирайте, что поместить на первые 50 страниц своей книги, потому что 90% читателей на этих 50 страницах и остановятся, даже если книга очень хорошая. Большинство людей прочитывает первую часть «Структурного программирования», написанную Дейкстрой. Некоторые читают вторую часть, написанную Хоаром. И, думаю, немногие добрались до третьей части, написанной Оле-Йоханом Далем при поддержке Хоара и названной «Иерархические структуры программ». В действительности это было введение в Симулу и ООП. Я был серьезным студентом: мне было сказано прочесть эту книгу, и я прочел ее от начала и до конца. Мне понравились первая и вторая части, а третью я счел разъяснительной.

Этим также объясняется, почему, когда через несколько лет вышло на сцену ООП и большинство заявляло, что оно заменит структурные методы, эти заявления казались мне бессмысленными. ООП с самого начала было частью структурных методов. Структурное программирование относилось к более низкому уровню, а объектно-ориентированное – к более высокому, но никакой пропасти между ними не было. Прочтя текст Даля и Хоара, я понял, что именно так нужно программировать. Когда в середине 1970-х я начал работать в компьютерной индустрии, мне повезло, что начальник разрешил мне купить компилятор Симулы, потому что он был весьма дорог, но очень хорош. Я активно использовал его в той фирме, где работал, и написал довольно интересные программы. Мне было совершенно очевидно, что других разумных способов писать программы нет, хотя многие считали меня чокнутым. В то время с объектно-ориентированным программированием было очень мало ясности.

В середине 1970-х, сразу после окончания университета, я со своим другом Клодом Бодуэном (Claude Baudoin) написал на французском языке книгу «Méthodes de Programmation» (изд. Eyrolles) – в некотором роде компендиум всего, что мы знали, чему научились в Стэнфорде и других местах. Книга пользовалась большим успехом. Фактически она печатается до сих пор, что несколько странно для книги 1978 года выпуска. Она стала учебником, без преувеличения, для пары поколений французских программистов – и русских, потому что была переведена тогда на русский язык в Советском Союзе[1]. В России она тоже имела большой

[1] Мейер Б., Бодуэн К. «Методы программирования».– Пер. с англ. – М.: Мир, 1982, т. 1, 2.

успех: бывая в России, я до сих пор встречаю людей, которые говорят, что изучали по ней программирование. Для объяснения приемов программирования, алгоритмов и структур данных там использовался псевдокод.

Я показал книгу Тони Хоару, и он сказал, что хотел бы включить ее в свою знаменитую серию по компьютерным наукам, издаваемую Prentice Hall, в английском переводе. Я согласился, и тогда он сказал: «Ты же говоришь немного по-английски, так почему бы тебе самому не сделать перевод?» Я сдуру согласился, вместо того чтобы найти того, кто сможет ее перевести. Это было самой большой глупостью в моей жизни: естественно, переводя книгу, я ее переделывал, потому что с момента первой публикации прошло уже три или четыре года. Я набрался опыта, и у меня возникли новые идеи. Я назвал книгу «Методология прикладного программирования», и она никогда не была опубликована, потому что я не смог ее завершить. Я переделывал каждое предложение. Это было очень непродуктивно, но при этом я улучшил псевдокод, которым пользовался в первой книге. В частности, я почувствовал, что не могу правильно описать программы или алгоритмы без контрактов. Нотация Eiffel для контрактов началась с этой работы.

Было еще одно важное событие. Я работал в промышленности, но взял академический отпуск в Калифорнийском университете в Санта-Барбаре (UCSB) и как гостю мне дали некоторые курсы, которые никто не хотел читать. Там были последовательные курсы 130A и 130B, «Структуры данных и алгоритмы», игравшие очень интересную роль, потому что в действительности у них было три цели. Официальной целью было преподавание структур данных и алгоритмов. Но были еще две скрытые и важные цели. Во-первых, курс должен был быть достаточно труден, чтобы отсеять изрядное количество студентов и оставить тех, которых стоило обучать компьютерным наукам. Во-вторых, в этом курсе студенты должны были изучить Си, потому что знание Си требовалось для других курсов.

Это был полный абсурд, потому что как бы ни был хорош Си, он плохо подходит для описания алгоритмов, не говоря уже об их изучении. Впечатление было ужасное, потому что вместо преподавания того, о чем я хотел рассказать в этом курсе, мне приходилось помогать студентам устранять ошибки в их программах, связанные с указателями Си и аналогичными его особенностями. Из этого я сделал два вывода. Во-первых, никогда больше не соприкасаться с Си в качестве языка, используемого человеком. Си – неплохой язык, если его генерирует компилятор, но идея писать на нем вручную совершенно абсурдна. Во-вторых, я понял, что единственный способ представить основные структуры данных и алгоритмы – это оснастить их в полной мере инвариантами циклов, вариантами циклов, пред- и постусловиями. Отчасти

поэтому, когда в конце того года мне нужно было разработать нотацию для нашей собственной работы в компании, которую мы тогда только что основали, я выбрал язык, которым хотел бы воспользоваться в том курсе, который читал в UCSB. Если шире, то это был результат изучения обширной литературы, поглощенности в течение многих лет работой по развитию программной инженерии, которую вели Даль, Дейкстра, Хоар, Вирт, Харлан Миллз, Дэвид Гриз, Барбара Лисков, Джон Гуттаг, Джим Хорнинг и подобные им люди, и, по сути, отслеживания эволюции языков программирования. Для меня выбор был очевиден. По существу, можно сказать, что Eiffel был спроектирован – я хотел сказать «в течение одного дня», но даже это будет неверно. Eiffel был спроектирован за 15 минут. Все было абсолютно очевидно.

Это Eiffel привел вас к идее контрактного программирования?

Бертран: Нет, путь был скорее обратным. То есть, концепция сложилась раньше. Язык только отражает ее. Для меня это бессмысленный вопрос, точнее, его нужно задавать тем, кто не пользуется контрактным программированием, – это они должны ответить, почему. Я просто не понимаю, как можно писать какие-то элементы программы, не потрудившись обрисовать, для чего эти элементы нужны. Это вопрос, который нужно задать Гослингу, Страуструпу, Алану Кэю или Хейлсбергу. Как они могут писать программы или разрабатывать язык, с помощью которого люди будут писать программы, и не обеспечить такой механизм? Просто не понимаю, как можно без этого написать хотя бы пару строк кода. Спрашивать человека, зачем он применяет контрактное программирование, – это все равно что спрашивать, зачем он пользуется арабскими цифрами. Пусть те, кто использует для умножения римские цифры, объясняют, зачем они это делают.

Я слышал, что контрактное программирование в ОО-языке реализует принцип подстановки Лисков. Вы согласны с этим?

Бертран: Я никогда не понимал, что такое принцип подстановки Лисков. Мне видится, что это просто полиморфизм.

Пожалуй, с этим я согласен, но загвоздка в том, что возможность подстановки должна быть полной. Нельзя, например, ограничить наследуемый тип так, чтобы он делал меньше, чем его родитель. По существу, вы должны выполнить тот же контракт, что и родительский класс.

Бертран: Мне кажется, это то, что появилось в Eiffel в 1985 году, – идея ослабить предусловие и усилить постусловие при переопределении подпрограммы. Если в этом состоит принцип подстановки Лисков, то я, видимо, согласен. Но в Eiffel это было до Барбары Лисков.

Мне нравится мысль о сходимости открытий.

Бертран: Часть работы Барбары Лисков, на которой мы непосредственно основываемся, – это понятие абстрактного типа данных из фундаментальной работы 1974 года. Конечно, была еще работа по языку CLU в MIT, также оказавшая большое влияние. Но принцип подстановки Лисков никогда не казался мне каким-то открытием.

Как контрактное программирование помогает в работе команды разработчиков?

Бертран: Оно позволяет участникам команды знать, чем занимаются их коллеги, не требуя вникать в то, как они это делают. Это дает возможность получать мгновенные снимки продуктов всех команд, исходя из одной лишь спецификации и не привязываясь к конкретному выбору представления. Оно также очень удобно для менеджеров.

Не возникает ли опасность чрезмерной определенности решения?

Бертран: Нет, на самом деле нет. Риск всегда в недостаточности определенности. Контракты редко бывают связаны с излишней специфицированностью. Такой риск возникает, если слишком рано начинают реализацию, покидая уровень спецификаций, но с контрактами этого случиться не может, потому что они описывают намерения, а не реализацию. Проблема спецификаций на основе контрактов противоположная: слишком многое остается недосказанным, потому что трудно специфицировать все полностью.

Я читал, что при использовании контрактов код не должен проверять условия контракта. Вся идея в том, чтобы затруднить возможный отказ кода. Не можете ли вы пояснить это решение?

Бертран: Видимо, многие претендуют на применение принципов контрактного программирования, не осмеливаясь применять это правило. Идея очень проста. Она касается предусловий. Если у вас есть предусловие для процедуры, в котором сказано «вот условия, которые нужно выполнить», то код самой процедуры не должен проверять контракты: вся ответственность за проверку контрактов на этапе исполнения – исходя из того, что в клиентах могут быть ошибки и они не обеспечат выполнение предусловия, – осуществляется в каком-то другом месте. Она возлагается на некий автоматический механизм, который будет использован во время тестирования и отладки. Но если у вас в коде есть и предусловие, и проверка этого условия и что-то неверно, то вы делаете одно и то же дважды; это значит, что вы не можете решить, на кого возложить ответственность за соблюдение условия, ограничения – на клиента или поставщика. Это действительно проверка того, что здесь применяется на самом деле, – контрактное программирование или какая-то разновидность защитного программирования? Готовы ли вы убрать проверки? Мало у кого хватает духа это сделать.

В контрактном программировании есть очень четкое правило, согласно которому предусловие – это ограничение, налагаемое на клиента, на того, кто осуществляет вызов, поэтому если нарушено предусловие, в этом виноват клиент. Подпрограмма за это не отвечает. Что касается постусловия, то за него отвечает поставщик, процедура. Если у вас есть предусловие, подразумевающее ответственность вызывающего, но затем его проверяет сама процедура, значит, вы в нерешительности и собираетесь написать массу бесполезного кода. Это, конечно, очень опасно, особенно потому, что этот код часто не пройдет процесс тестирования и отладки во время разработки. На самом деле, это лишь вопрос серьезного отношения к спецификации.

Насколько важно различие между спецификацией и реализацией?

Бертран: Это действительно хороший вопрос. Это различие очень существенно, но оно относительно. То есть абсолютно невозможно сказать, что это абсолютная спецификация или что это абсолютная реализация. Одна из особенностей ПО заключается в том, что какой бы программный элемент вы ни взяли, он будет спецификацией для чего-то более конкретного и реализацией для чего-то более абстрактного. Возьмите даже конструкцию, которая выглядит совершенно как реализация, скажем присваивание := A+1 или A := B. Большинство скажет вам, что это чистая реализация. Но если вы пишете компилятор, то она для вас станет спецификацией, которая будет развернута в десяток команд машинного языка или Си. Различие важно, но что действительно сложно в ПО, так это то, что при достаточно большом размере для определенного масштаба технология написания реализации становится очень похожей на технологию написания спецификации.

Например, все, кто пишет формальные спецификации, сталкиваются с поразительным явлением: когда пишешь достаточно крупную формальную спецификацию, приходишь к тому, что делаешь вещи и задаешь себе вопросы, удивительно похожие на то, что делаешь и о чем себя спрашиваешь, когда пишешь настоящие программы. Поэтому разница всегда относительна. Причина в том, что в программировании мы не работаем с физическим материалом. Мы не работаем с конкретными, ощутимыми, материальными элементами. Все, с чем мы работаем, – это абстракции, поэтому разницу между реализацией и спецификацией, по существу, составляет один уровень абстрагирования. Поэтому обычно бессмысленно указывать, что нечто является реализацией, а не спецификацией. Но можно сказать, что *X* является спецификацией *Y*; или, по-другому, *Y* является реализацией *X*. Это утверждение, которое имеет смысл: его можно опровергнуть. Но утверждения «*X* является спецификацией» или «*X* является реализацией» не могут быть опровергнуты. На них нельзя точно ответить «да» или «нет».

Как язык программирования связан с конструкцией программ на нем?

Бертран: Одна из действительно уникальных сторон Eiffel – одно из очевидных свойств ПО, которое почти никто не считает ни очевидным, ни просто верным, – это полная непрерывность связи между идеей и реализацией. Мы называем это цельностью разработки (seamless development), и это, возможно, самый важный аспект Eiffel. Все остальное направлено на его поддержку. Из этой идеи, например, следует, что между проектом и реализацией фактически нет разницы. Реализация, если перефразировать известное выражение, – это ведение проектирования другими средствами. Разница только в уровне детализации и уровне абстракции. В частности, Eiffel в той же мере предназначен служить языком анализа и проектирования, в какой быть языком реализации. В общем и целом, это, по существу, метод, а не язык, но та его часть, которая является языком, служит как для анализа и проектирования, так и для реализации.

Те, кто работают с Eiffel, обычно не пользуются UML и подобными ему средствами, которые Eiffel-разработчику представляются больше разговорами, имеющими мало пользы для программирования. Eiffel – это инструмент, предназначение которого – помочь вам, – вы говорите о проектировании, но я бы сказал, что сначала на уровне спецификаций и анализа, затем при проектировании и потом в реализации, – поддержать вас в этом процессе. Но принципиальной разницы между этими задачами нет – согласно моим представлениям и представлениям разработчиков Eiffel в целом.

Еще нужно отметить, что язык должен быть как можно скромнее. Сегодня есть много языков, которые я называю высокомерными, с массой необычных символов и условностей, которые нужно изучить, чтобы быть допущенным во внутренний круг. Например, многие из доминирующих сегодня языков фактически происходят от Си. Они стали результатом ряда добавлений и исключений из Си, и чтобы овладеть ими, нужно разобраться с большим количеством багажа.

Идея Eiffel – не стану утверждать, что Eiffel лишен всякого багажа, но все же его немного – в том, что если вы проектируете, то думаете о проекте, а не о языке. Лучшей похвалой, услышанной мною от пользователей Eiffel, были их слова о том, что, используя этот язык, они думают только о своей задаче и ни о чем больше. Это лучшее влияние, которое язык может оказать на проектирование.

Не пробовали ли вы искать другие решения помимо действий с объектами на уровне языка? Скажем, компоненты типа мелких утилит UNIX, из которых можно строить сложные функции с помощью конвейеров. В конце концов, если вы пишете кому-то письмо и вам нужно

что-то описать, вы же не пользуетесь французскими, итальянскими, английскими или какими-то еще объектами?

Бертран: Конечно, механизмы UNIX очень элегантны, но они слишком мелки для той работы, которая нужна для создания большой программы. Мой опыт показывает, что объекты – это единственный механизм, доказавший возможность своего масштабирования для крупных систем. Единственный другой подход, который я, возможно, стал бы рассматривать, – это функциональное программирование, но я не уверен в его пригодности. Идея заманчивая, очень элегантная, и у нее можно многому поучиться, но на высшем уровне она проигрывает в сравнении с объектами. Объекты – или следует говорить «классы» – значительно эффективнее для описания крупномасштабной структуры системы.

Аналогию нужно проводить не столько с естественными языками, сколько с математикой. Кроме того, это скорее классы, чем объекты. Классы – это не что иное, как перенос в программирование понятия структуры, успешно действующего в математике: группы, поля, кольца и так далее. То есть математика берет объекты, природа которых может быть очень различной, например числа и функции, и показывает, что в обоих случаях у вас одинаковая структура, определяемая операциями с одинаковыми свойствами. Затем вы извлекаете из этого одно абстрактное понятие, скажем группы, моноида или поля. Эта идея успешно применяется в математике последние 200 лет. В этом отношении классы, или объекты, являются не столь уж новым понятием. Это прямой перенос обычного понятия математической структуры.

Многие современные системы состоят из компонентов, размещенных в сети. Должны ли в языке быть отражены такие сетевые аспекты?

Бертран: Это желательно, и в Eiffel можно это делать, но я бы не стал утверждать, что Eiffel в этом сильно преуспел. Сейчас ведется множество разработок в области динамического обновления и параллелизма, результаты которых мы вскоре увидим, но пока их нет. Думаю, это важный вопрос. Можно спорить, к чему он относится – к языку или реализации, но какая-то поддержка в языке необходима.

Какую вы видите связь между объектно-ориентированной парадигмой и параллелизмом?

Бертран: Думаю, параллельная обработка данных имеет чрезвычайно большое значение. Я много об этом писал. В частности, есть много литературы, посвященной SCOOP – разработанной нами модели объектно-ориентированного параллельного программирования. Суть в том, что элементарные подходы оказываются недейственными. Довольно часто говорят, что параллелизм и объекты – это почти одно и то же, это должно замечательно работать, объекты параллельны по своей сути, и счи-

тают, что и делать ничего не нужно. Все совершенно не так. Объединить идеи объектной ориентированности с параллелизмом на элементарном уровне не получится.

Скажу несколько слов о SCOOP. В основе лежит понимание того, что стандартное понятие контракта не может одинаково интерпретироваться в параллельном и в последовательном контекстах. Идея SCOOP в том, чтобы взять модель последовательного ООП и построить для нее минимальное расширение, которое будет поддерживать параллелизм. Этот путь весьма отличен от тех, которыми идут все остальные. Например, если взять исчисление процессов, то там выбран прямо противоположный подход: выяснить, какая система параллельности лучше, а потом устроить поверх нее программирование. Получается нечто, сильно отличающееся от обычных способов программирования. SCOOP учитывает, что человеку трудно думать параллельным образом, последовательно он думает гораздо эффективнее, поэтому SCOOP стремится скрыть сложность параллельности в реализации, в модели. В результате появляется возможность программировать параллельным способом, который очень близок к последовательному программированию и позволяет сохранить привычный образ мышления.

На каком уровне нужно решать проблему параллелизма? Например, JVM почти прозрачным образом управляет некоторыми ее аспектами.

Бертран: Очевидно, потоки Java очень удобны во многих приложениях, но они не очень хорошо согласуются с объектно-ориентированной природой вещей. По сути, это семафоры в понимании Дейкстры. На самом деле, в Eiffel есть библиотека EiffelThreads, которая делает примерно то же самое. Думаю, каждому должно быть ясно, что такие решения годятся на какое-то время, но не масштабируются. Слишком много сохраняется возможностей для возникновения гонки и взаимной блокировки. Задача в том, чтобы программисты были автоматически защищены от таких проблем, а это требует работы на более высоком уровне описания. Как вы сказали, это означает, что все большую часть работы нужно переложить на реализацию.

Многократное использование и универсальность

Как в Eiffel решается проблема модификации и развития программ?

Бертран: Расширяемость, наряду с многократным использованием, с самого начала была одной из наших главных целей. До некоторой степени это причина продолжения работы над Eiffel, потому что, как я говорил, мы сначала разрабатывали Eiffel как инструмент для внут-

реннего использования, а не как товар на продажу. К тому, чтобы пересмотреть нашу позицию и сделать Eiffel более широкодоступным, нас привели высказывания разработчиков о том, что в отличие от тех языков, которыми они пользовались раньше, им теперь гораздо проще менять свои решения, не страдая от своей нерешительности.

Думаю, здесь важны несколько аспектов. Во-первых, в Eiffel тщательно скрывается информация, чтобы одни модули не видели изменений в других. Для потомков информация не скрывается, потому что в этом нет смысла, но от клиентов скрывается. Например, совершенно поразительно и несколько неприятно видеть, что в новейших объектно-ориентированных языках все еще можно непосредственно присваивать значение атрибуту – полю объекта. В Eiffel вам такое не позволено, потому что это нарушает правило скрытия информации. Для программ это катастрофа.

Далее, механизм наследования очень гибок и позволяет писать программы путем модификации существующих шаблонов. Универсальность создает еще один уровень гибкости.

Отсутствие в языке механизма, который выше класса, дает возможность комбинировать классы очень гибким образом. Контракты также играют положительную роль, потому что когда вы вносите в программу изменения, очень важно знать, что́ вы меняете, в частности, касаются ли изменения спецификации или только реализации. При внесении в программу изменений вы должны решить, будут ли эти изменения чисто внутренними, не затрагивающими контрактов, – и тогда вы знаете, что они никак не повлияют на клиентов, – или же изменения коснутся также контрактов, и тогда, конечно, нужно в точности знать, каким образом. Тем самым устанавливается уровень детализации для контроля размера изменений.

Как должны относиться к многократному использованию разработчики? Я задаю этот вопрос, потому что ряд тех, у кого я брал интервью, по сути сказали, что даже если вы строите классы, об их повторном использовании лучше позабыть, потому что оно требует слишком много труда, и что задумываться о многократном использовании стоит лишь тогда, когда обнаружится, что вы регулярно используете некий класс в контекстах разного типа. Вот тогда и стоит потратить лишнее время, сделав класс многократно используемым.

Бертран: Думаю, это относится к тем случаям, когда вы не сильны в вопросах многократного использования или вообще новичок. Верно, если у вас нет опыта повторного использования, то при попытке сделать свою программу более общей, чем того требуют текущие технические требования, вы потратите массу времени и можете не достичь успеха. Но я утверждаю, что если вы освоили повторное использование, если

у вас большой опыт работы с повторно используемыми компонентами других разработчиков и есть собственный опыт создания повторно используемого ПО, то вы успешно справитесь с задачей.

Мне кажется, что многие допускают ошибку, поскольку не понимают, что в повторном использовании есть два аспекта и один должен предшествовать другому. Это аспект потребления и аспект изготовления. Как потребитель, вы просто используете существующее ПО в собственных приложениях; многие делают это в основном для экономии времени. В аспекте изготовления вы делаете свое собственное ПО более удобным для повторного использования. Если вы решите сразу стать производителем повторно используемого ПО, вас ждет неудача, потому что такое производство действительно требует специальных технологий. Вы потратите массу времени, делая свою программу более универсальной, но вам придется гадать, в каком направлении она может быть обобщена позднее, и ваши догадки окажутся, скорее всего, неверны.

Однако если начать с более скромной позиции потребителя – изучить высококачественные библиотеки для многократного использования, способы их создания и проектирования, имеющиеся у них API, – тогда можно применить изученный вами стиль к своим собственным программам. Нечто очень похожее есть в мире Eiffel. Люди учатся программировать на Eiffel, изучая стандартные библиотеки, такие как EiffelBase или графическая библиотека EiffelVision. Это библиотеки высокого качества, которые могут служить моделью для хороших программ. Изучив их, вы сможете применить те же принципы в собственных программах и сделать их гораздо лучше, в частности более пригодными для многократного использования. Вот в таком направлении нужно идти: начать как потребитель и на основе полученного опыта стать производителем. Мой опыт показывает, что этот путь может быть успешным. Эту мысль я развил в своей книге «Reusable Software» (Повторное использование программного обеспечения), изд. Prentice-Hall, 1994.

При таком подходе можно сделать свое ПО многократно используемым. С точки зрения ускоренного и экстремального программирования о повторном использовании заботиться не нужно, потому что это пустая трата времени. Думаю, это относится только к тем, кто не умеет писать многократно используемые программы, потому что не потрудился изучить по хорошим образцам, как это правильно делается.

Возможно, это также зависит от используемого языка программирования.

Бертран: Безусловно зависит – тут меня не нужно долго уговаривать. При проектировании Eiffel решалось три основных задачи. Первой была, конечно, корректность или, более общо, надежность. Второй была расширяемость, простота модификаций, а третьей – многократ-

ное использование кода. Поэтому повторное использование учтено всюду. Например, совершенно поразительно, что обобщенные классы у нас появились с самого начала. Это было совершенно необходимо для многократного использования, но долгие годы над нами постоянно смеялись. На первой конференции OOPSLA в 1986 году у нашей компании был стенд с соответствующим плакатом, и подходя к стенду люди смеялись, видя слово «genericity», которого, по их словам, просто нет в английском языке. Никто не понимал, о чем идет речь.

Через несколько лет в C++ появились шаблоны. Когда в 1995 году появилась Java, там не было генериков, и все говорили, что они не нужны, что это только запутывает объектно-ориентированные языки. Как бы то ни было, спустя десять лет генерики появились, но сделано это было сложным и, на мой взгляд, не вполне удовлетворительным способом – не столько из-за плохого проектирования, сколько из-за требований совместимости. Когда вышел C#, я не поверил глазам – генерики опять отсутствовали, несмотря на опыт, полученный с Java, но, конечно, обобщенные классы были добавлены – лет семь спустя.

Все это было предусмотрено в Eiffel с самого начала в расчете на многократное использование. Конкретные детали механизма наследования, конкретное сочетание переименования, переопределения, отмены определения, имеющиеся в Eiffel, механизм множественного наследования – все это оправдано и мотивировано многократным использованием. Конечно, важную роль играют контракты. Я уже сказал, что не понимаю, как можно программировать без контрактов, но что еще труднее понять – как могут существовать предположительно многократно используемые компоненты без четкой спецификации того, что эти предположительно многократно используемые компоненты делают.

Понемногу люди стали это понимать. Было объявлено, что в .NET 4.0 будут библиотека контрактов Code Contracts и все основные библиотеки; Mscorelib должна быть заново задокументирована и спроектирована с контрактами. Всего лишь через 23 года после Eiffel. Наконец-то начинают понимать, что не может быть многократного использования без контрактов. Потребовалось время. Разумеется, за это время Eiffel продолжал вводить новые идеи, чтобы оставаться лидером.

Когда и как вы поняли, что универсальность столь же важна, как классы?

Бертран: Это представление и понимание, как мне кажется, возникли скорее, в академическом контексте, а не вследствие потребностей производства. Я в своей профессиональной деятельности больше времени провел в промышленности, но немного потрудился и на академической ниве. Пару раз, в 1984 и 1985 годах, я читал в университете Санта-

Барбары курс «Современные концепции в языках программирования». Его содержание было весьма вольным.

Я хотел посмотреть, что происходило в то время на передовой языков программирования. Я включил в свой курс Аду, о которой много спорили, и Симулу, о которой никто не спорил, но я видел, что объектно-ориентированным идеям этого языка принадлежит будущее.

Эта проблема не могла не возникнуть при чтении данного курса, потому что одну неделю я рассказывал об универсальности, а другую – о наследовании, или наоборот. Естественно, возник вопрос, как эти вещи соотносятся между собой. Не помню, спросил ли об этом кто-то из студентов, хотя могло быть и так. Помню, как спрашивал себя сам: «Что же, я так и буду одну неделю доктором Универсальность, а другую неделю – мистером Наследование?»

Это вынудило меня задаться следующим вопросом: «Чего такого нельзя сделать с помощью одного, что можно сделать с помощью другого?» – и наоборот. Конечно, это было результатом многих предшествующих обсуждений и размышлений, но в обсуждении языков программирования общество разделилось на два лагеря: тех, кто считал, что Ада удовлетворяет все мыслимые потребности языков программирования в гибкости, и небольшую группу открывших для себя ООП и наследование.

В рабочих группах велись обычные споры: «На моем языке это получается лучше» – «Нет, на моем». Насколько могу судить, никто не сделал следующий шаг и не попытался выяснить, какая связь имеется между двумя механизмами и как точно сравнить силу их выразительности.

В своем курсе я провел для них некоторый сравнительный анализ. Потом поступило предложение представить доклад для первой конференции OOPSLA, и естественным было выбрать эту тему.

Я сел и написал, что думаю по этому поводу, и получился доклад «Универсальность и наследование» в трудах первой конференции OOPSLA.

Вы опубликовали этот доклад еще до того, как эта проблема была выявлена в наиболее распространенных ОО-языках. Даже в Smalltalk она никак не рассматривается.

Бертран: В Smalltalk это безразлично, потому что в Smalltalk есть динамическая типизация, и ничего этого не нужно. На самом деле, это было очень странно. Вспоминаю слова Шопенгауэра – что-то вроде «сначала они над вами смеются, а потом они...»

...Сначала они вас не замечают, потом они над вами смеются. Да.

Бертран: На самом деле, буквально так и происходило. На первой конференции OOPSLA я представил свой доклад под вывеской Универси-

тета, так что это была действительно научная работа, а у компании был также стенд, очень кустарный, потому что у нас совсем не было денег. Плакаты были самодельными, некоторые рукописными.

К нашему стенду подходили люди и, насмеявшись, приводили друзей, чтобы они тоже посмеялись. Как я уже говорил, одним из поводов для смеха для них было слово genericity (универсальность). Там были большие парни из HP – действительно большие, я не шучу. Некоторые пиджаки из HP подходили по несколько раз, приводя все новых приятелей, чтобы показать это слово и им. «Как это произносится? По-французски, что ли?» Они громко зачитывали его вслух на разные лады: «Наверно, это generisissyty!» – и так далее.

Таков был дух времени. А через 20 лет мне присылают на отзыв статьи, где говорится, что универсальность появилась благодаря Java. Жизнь удивительна.

Проверка языков

Вы, я слышал, говорите по меньшей мере на трех языках: английском, французском, конечно, и немецком. Ваша многоязычность повлияла на вас как на разработчика языков?

Бертран: Если коротко, то да. На самом деле, немецким я владею не очень хорошо. Мой родной язык французский. С английским стараюсь изо всех сил. Я довольно бегло говорю по-русски. У меня даже есть диплом магистра по русскому языку, хотя я говорю далеко не на том уровне, который должен быть у человека с дипломом магистра. Я прилично говорю по-итальянски. Я без особого труда читаю лекции по-русски. По-итальянски я могу читать лекцию в течение 15 минут, но потом у меня начинается перегрев мозга. Это было сказано для большей точности, но ответом на ваш вопрос определенно будет «да».

В компьютерную науку я попал частично из-за своего интереса к языкам. Понимание того, что одно и то же можно сказать по-разному, что нет взаимно однозначного соответствия, что к вещам можно подходить по-разному, что иногда правильным будет употребить существительное, а иногда нюансы лучше передает глагол, – все это оказало определенное влияние и сильно помогло мне. Я также полагаю, что тот, кто знает хотя бы один иностранный язык, лучше говорит на своем родном языке.

Кроме того, обычно во всяких технических начинаниях, в частности в программировании, вы не просто пишете программы, но и пишете на английском или каком-то другом естественном языке. Некоторые уси-

лия, потраченные на улучшение навыков письма или изучение одного или нескольких языков, хорошо окупаются.

Вы подходите к программированию с точки зрения математики или лингвистики либо как-то их комбинируете?

Бертран: Мне следовало бы подходить к программированию более математически, чем я это делаю. Я убежден, что через 50 лет программирование станет разделом математики.

Есть люди, уже давно продвигающие математический подход к программированию. На практике он не утвердился, за исключением отдельных областей, где строят небольшие, жизненно важные системы и другого выхода не остается. В конце концов, программирование – это действующая математика: математика, которая может быть интерпретирована машиной. Думаю, в будущем характер программирования станет еще более математическим.

Что касается моего личного подхода, то я охарактеризовал бы его как сочетание того, что называется лингвистическим подходом, более самопроизвольного, творческого, дискурсивного подхода, и старания соблюдать строгость и математичность. Конечно, в Eiffel влияние математики сильнее, чем в большинстве других существующих языков, исключая, пожалуй, функциональные, вроде Haskell.

Я спрашивал у многих разработчиков, как они относятся к тому, чтобы начать с маленького строгого базового языка и затем развивать его, – возьмите лямбда-исчисление. Можно организовать любые вычисления, если вы можете применять функции. Как вам нравится такой подход?

Бертран: Не думаю, что он очень удачен. Проблема программирования в том, чтобы сочетать науку и технику. С одной стороны, программирование существенно опирается на науку, и, как я сказал, в принципе, программирование – это математика. Но другая сторона – техническая. Некоторые из имеющихся сегодня программ сложнее едва ли не любого артефакта, когда-либо созданного человечеством. Размер больших операционных систем, таких как дистрибутивы Linux, Vista, Solaris, исчисляется десятками миллионов строк кода, иногда больше 50 миллионов. Это невероятно сложные инженерные конструкции. Многие из проблем, с которыми они сталкиваются, являются по существу инженерными проблемами.

Разница между наукой и техникой, если несколько упростить дело, состоит в том, что в науке вам нужно лишь несколько толковых идей; в технике приходится заботиться об огромном количестве деталей, большинство из которых не очень сложны, но их очень много. Таким

образом, с одной стороны немного остроумных идей, а с другой – множество не очень сложных вещей. Для программирования характерно, что требуется и то, и другое. Может показаться, что я противоречу тому, что только что сам сказал, а именно, что через несколько десятилетий программирование станет по существу математическим, но, думаю, здесь нет противоречия. Объясню почему.

В основе своей программирование – это всего лишь математика, но нужно обратить внимание на слова *в основе*. На практике программирование включает в себя также всякие инженерные задачи, которые приходится решать. Если вы пишете операционную систему, то должны гарантировать, что тысячи драйверов для различных устройств, написанных неискушенными программистами, не вызовут краха вашей операционной системы. Вам нужно позаботиться о том, чтобы диалоговые окна, задействованные в работе, использовали многочисленные естественные языки. Нужно создать очень сложный набор механизмов для пользовательских интерфейсов: основные идеи могут быть просты, но детали неисчислимы.

Сложность программирования имеет двойственный характер: это научная сложность и техническая сложность. Наличие очень сильной математической основы, например лямбда-исчисления, поможет вам справиться с первой частью, но не со второй, и поможет с той частью, которая на сегодняшний день лучше всего понята. Лямбда-исчисление очень хорошо подходит для моделирования базовой части языка программирования на уровне, скажем, Паскаля или Лиспа, но современные языки программирования гораздо более претенциозны.

В конечном счете, мы должны все свести к очень простым математическим принципам, но одних математических принципов недостаточно для решения задач, встающих при создании крупных современных программ.

Это то, что отличает академические языки программирования от промышленных языков программирования?

Бертран: Совершенно верно. Когда создавалась Ада, этот язык критиковали за огромный размер и сложность, и его создатель Жан Ишбиа сказал в одном интервью: маленькие языки решают маленькие задачи. В этом есть известная доля правды. Думаю, он, в сущности, отвечал на критику таких людей, как Вирт, который глубоко верил, что «малое прекрасно», но он прав и в целом.

Есть ли что-то между структурным программированием и ООП? Вы сказали, что считаете структурное программирование хорошим способом организации маленьких программ, а ООП – хорошим способом

организации больших программ. Попадает ли какая-то часть программ по размеру в промежуток между тем и другим?

Бертран: Нет, я бы не стал пользоваться ничем, кроме ООП. В принципе, я изучал то и другое одновременно и не вижу причин использовать в своих разработках технологии не-ООП, за исключением, может быть, каких-то маленьких одноразовых скриптов. «Объектная ориентированность» означает просто применение к программам математического понятия структуры, против чего нет веских аргументов.

Программы бывают либо маленькими, либо большими.

Бертран: Нет веских причин не использовать классы. Я точно не знаю, что об этом думал Дейкстра. Он не был большим сторонником ООП, но мне также неизвестно, чтобы он его критиковал: когда ему что-то не нравилось, он мог высказаться очень откровенно и резко.

Вы сказали, что в Eiffel вам был очень нужен механизм поточной сериализации. Можно поинтересоваться зачем?

Бертран: Первым сделанным нами приложением был, как я уже сказал, Smart Editor, выпущенный под торговым названием ArchiText. От редактора всегда требуется возможность работы с небольшой структурой данных, находящейся в памяти, с ее последующим сохранением. Можно каждый раз проводить парсинг текста, а потом обратную операцию, но это абсурдный способ. Допустим, вы редактируете текст и восстановили для текста небольшую структуру, и у вас есть абстрактное синтаксическое дерево или другое эффективное внутреннее представление: зачем вам превращать его в текст, а в следующий раз заново создавать это дерево? Нужно постоянно использовать эту абстрактную структуру. Если нужно сохранить ее, нажмите кнопку, и поточный механизм все сделает за вас.

Так было с первым приложением, но и все последующие приложения оказывались такими же. Если вы пишете компилятор, происходит то же самое. Допустим, ваш компилятор выполняет несколько проходов. Каждый проход берет структуру, полученную при предыдущем проходе, декорирует ее, немного манипулирует с данными и записывает результат на диск. Нет надобности каждый раз делать эту запись каким-то особым способом. Нужно просто нажать на кнопку. Есть десятки приложений, в которых требуется такая вещь.

Можно вставлять промежуточные этапы.

Бертран: Верно. Вас не ограничивает какая-то одна конкретная структура обработки.

Вы использовали выражение «цельность разработки» и сказали, что это одна из принципиальных идей в Eiffel. Что это такое?

Послесловие

Мое удовольствие от работы над этим проектом можно описать одним словом – восторг. В каждой беседе можно почерпнуть глубокие знания, исторические сведения и практическое понимание предмета. Заразительным также оказался энтузиазм интервьюируемых по поводу проектирования языка, его реализации и дальнейшего развития.

Например, Андерс Хейлсберг и Джеймс Гослинг снова заставили меня восторгаться C# и Java. Чак Мур и Эдин Фалкофф убедили меня заняться изучением Форта и APL – языков, придуманных еще до моего рождения. Ал Ахо соблазнил меня описать свой класс компиляторов. У каждого из тех, кто дал нам интервью, я почерпнул столько идей, что времени не хватит их осмыслить!

Я искренне благодарен этим людям не только за то, что они уделили нам с Федерико столько времени, но и за то, что они осветили нам всем путь к богатейшему полю новых открытий. Вот главные уроки, которые я извлек из этого предприятия:

- Простоту конструкции и реализации нельзя переоценить. Сложность всегда можно добавить потом. Мастер от нее избавляется.
- Отдавайтесь своей любознательности со страстью. Многие из величайших открытий и изобретений появились благодаря тому, что кто-то оказался в нужное время в нужном месте с правильным ответом наготове.
- Изучайте настоящее и прошлое своего любимого дела. Каждый из участников наших интервью работал с другими умными и трудолюбивыми людьми. Очень важно передавать знания друг другу.

Модные языки непрерывно сменяют друг друга, но те проблемы, с которыми сталкивался каждый из их пионеров, по-прежнему преследуют нас, и их решения по-прежнему актуальны. Как сопровождать ПО? Как найти лучшее решение задачи? Как удивить и порадовать пользователей? Как удовлетворить неугомонное стремление все переделать и в то же время сохранить работоспособность действующих решений?

Теперь я лучше знаю, как ответить на эти вопросы. Надеюсь, эта книга помогла вам в обретении собственной мудрости.

Шейн Уорден