

13

Программные архитектуры: объектно-ориентированные и функциональные

Бертран Мейер

Одним из аргументов в пользу функционального программирования считается улучшение модульности архитектуры. Анализ публикаций, поддерживающих этот подход (прежде всего на примере инфраструктуры для оформления финансовых контрактов), поможет нам оценить его сильные и слабые функционального программирования и сравнить его с объектно-ориентированным проектированием. Общий вывод будет таким: объектно-ориентированное проектирование (особенно в его современной форме с поддержкой высокоуровневых функциональных объектов, или «агентов») *замещает* функциональный подход, сохраняя большинство из его преимуществ, но при этом предоставляет высокоуровневые абстракции, лучше приспособленные для возможного расширения и повторного использования.

Обзор

«Красота» как лозунг программной архитектуры не является сугубо субъективным показателем. Существуют четкие критерии (Meyer, 1997):

Надежность

Оказывает ли архитектура положительное влияние на правильность и устойчивость работы программного продукта?

Расширяемость

Насколько легко она адаптируется к изменениям?

Универсальность

Имеет ли решение общий характер – или еще лучше, возможно ли преобразовать его в *компонент*, который может быстро и напрямую подключаться к новому приложению?

Успех объектных технологий в значительной степени обусловлен тем заметным положительным влиянием, которые они могут оказать (при правильном применении на методологическом уровне, а не простым выбором объектно-ориентированного языка программирования) на надежность, возможность расширения и универсальность создаваемых программ.

Методология *функционального программирования* старше объектно-ориентированного подхода. Ее истоки уходят к языку Lisp, который существует уже почти 50 лет. Те счастливицы, которые изучали ее на ранней стадии, всегда будут помнить ее, словно первый поцелуй, сладость и предчувствие еще больших наслаждений. В последнее время функциональное программирование переживает новый подъем в связи с появлением таких языков, как Scheme, Haskell, OCaml и F#, сложных систем типизации и нетривиальных языковых механизмов (таких как монады). Иногда функциональное программирование даже представляется как следующий шаг после объектно-ориентированных методологий. В данной главе мы сравним эти два подхода по критериям программной архитектуры, указанным выше. Оказывается, объектно-ориентированная архитектура, особенно обогащенная новейшими достижениями (такими, как *агенты* в терминологии Eiffel, также называемыми «замыканиями» или «делегатами» в других языках), превосходит функциональное программирование, сохраняя его архитектурные преимущества и исправляя недостатки.

Для адекватной оценки полученных результатов важно понимать как ограничения анализа, так и аргументы, опровергающие некоторые из них. Ограничения:

Малый объем выборки

Анализ в основном базируется на двух примерах функционального проектирования. Это обстоятельство может поставить под сомнение общий характер выводов.

Малая детализация

Примеры, представленные в главе, позаимствованы из статьи (Peyton Jones et al., 2000) и презентации PowerPoint (Eber et al., 2001), которые в дальнейшем будут именоваться «статьей» и «презентацией» (с дополнениями из классической статьи по функциональному программированию [Hughes, 1989]), в презентации могут быть опущены некоторые подробности, присутствующие в более подробном документе).

Ограниченность подхода

Мы ограничимся рассмотрением аспектов модульности. К сильным сторонам функционального программирования также относятся и другие критерии – например, элегантность декларативного подхода.

Субъективизм экспериментатора

Автор настоящей главы в течение долгого времени являлся сторонником объектных технологий и вносил активный вклад в их развитие.

Возможная критика отчасти компенсируется следующими обстоятельствами.

- Примеры функционального программирования позаимствованы из реальной жизни, а именно из практики компании, бизнес которой основан на применении методов функционального программирования. Приведенный пример – определение сложных средств для описания финансовых контрактов – отражает сложные проблемы, с которыми сталкивается финансовая отрасль и которые недостаточно хорошо решаются текущим инструментарием (по утверждениям автора, являющегося экспертом в этой отрасли). Возникает предположение, что такая ситуация типична для нынешнего состояния дел (первый пример этой главы – описание пудингов – имеет чисто академический характер и приводится исключительно в учебных целях).
- Один из авторов статьи (С. Пейтон Джонс), также упоминаемый в презентации как соавтор теоретической работы, является ведущим проектировщиком языка Haskell и одной из самых заметных фигур в области функционального программирования, что придает его утверждениям достоверность. Статья, использованная в качестве

ве дополнительного примера в разделе «Оценка модульности функциональных решений», считается в высшей мере авторитетной и была написана одним из ведущих экспертов сообщества функционального программирования (Дж. Хьюз).

- Невзирая на все замечания, описанные в этих документах решения весьма элегантны и, несомненно, появились в результате серьезных размышлений.
- В примерах не задействована концепция изменяемого *состояния*, благоприятствующая применению объектно-ориентированных языков программирования.

Следует заметить, что такие механизмы, как агенты, являющиеся исключительно важными компонентами объектно-ориентированных решений, явно вдохновлены идеями функционального программирования. Таким образом, наше заключение ни в коей мере не отрицает вклада функциональной школы, а всего лишь отмечает, что объектно-ориентированный (ОО) стиль лучше подходит для определения общей архитектуры надежных, расширяемых и универсальных программ, а среди структурных блоков таких архитектур могут присутствовать комбинации как методов ОО, так функциональных методов.

Еще несколько замечаний по поводу следующего обсуждения.

- Объектная технология, использованная в примерах, представлена на языке Eiffel. Мы не пытались анализировать, что произойдет при *удалении* таких механизмов, как множественное наследование (не поддерживаемое в Java и C#), параметризация (отсутствующая в ранних версиях этих языков), контракты (отсутствующие за пределами Eiffel, кроме JML and Spec#) и агенты с их аналогами (отсутствуют в Java) или при *добавлении* таких механизмов, как перегрузка и статические функции, противоречащих изначальной простоте объектно-ориентированного подхода.
- Темой обсуждения являются архитектура и проектирование. Несмотря на свое название, функциональное программирование (как и объектные технологии) имеет отношение к этим задачам и не ограничивается «программированием» в ограниченном смысле, т. е. реализацией. Методология Eiffel явным образом вводит континуум из спецификации в проектирование и реализацию через концепцию плавной разработки. Мы не будем сколько-нибудь подробно обсуждать аспекты обеих методологий, относящиеся к реализации.
- Также для практического программирования важны аспекты выразительности и удобства записи. Они принимаются во внимание в той степени, в которой влияют на ключевые критерии архитектуры и проектирования. Однако в общем и целом обсуждение будет

относиться не к синтаксической форме, а к семантическому содержанию.

И еще два предварительных замечания. Первое касается терминологии: по умолчанию термин «контракт» будет использоваться для обозначения финансовых контрактов, относящихся к предметной области статьи и презентации; не путайте их с концепцией контрактов в программировании (Meyer, 1997), относящейся к элементам спецификации (предусловия, постусловия, инварианты). В случае возможной неоднозначности будут использоваться термины *финансовые контракты* и *программные контракты*.

Второе замечание больше напоминает самооправдание: когда во второй половине обсуждения переходит в объектно-ориентированную область, текст содержит больше ссылок и цитат из предыдущих публикаций автора, чем допускают приличия. Дело в том, что широкое распространение объектных технологий сопровождалось утратой некоторых неочевидных, но (по нашему мнению) критически важных принципов, как, например, разделение команд и запросов (см. раздел «Проблема состояния» далее в этой главе); по этой причине короткие напоминания необходимы. За полными обоснованиями обращайтесь по ссылкам.

Примеры

Общая цель статьи и презентации заключается в создании удобного механизма описания финансовых контрактов и работы с ними — особенно с современными финансовыми инструментами, которые бывают очень сложными, как в следующем примере из презентации (в числовых значениях доносятся ностальгические отзвуки тех времен, когда основные валюты находились в несколько иных соотношениях):

«Против обещания оплаты USD 2.00 27 декабря (по цене опциона) владелец имеет право 4 декабря выбрать одно из двух:

- получить USD 1.95 29 декабря или
- иметь право 11 декабря выбрать одно из двух:
 - получить EUR 2.20 28 декабря или
 - иметь право 18 декабря выбрать одно из двух:
 - получить GBP 1.20 30 декабря или
 - немедленно доплатить EUR 1.00 и получить EUR 3.20 29 декабря».

Приведенные в этом разделе цитаты в кавычках взяты непосредственно из презентации или статьи. Элементы, не заключенные в кавычки, — наши интерпретации и комментарии.

Презентация начинается с учебного примера, который помогает проиллюстрировать материал: контракты заменяются *пудингами*. По точному описанию пудинга повар должен иметь возможность «вычислить содержание сахара», «оценить время приготовления» и получить «инструкции по приготовлению». «Плохое решение» выглядит так:

- «Перечислить все разновидности пудингов (со взбитыми сливками, лимонный, голландский яблочный, рождественский).
- Для каждого пудинга записать содержание сахара, время приготовления, инструкции и т. д.».

Хотя презентация не объясняет, почему этот подход плох, причины легко понятны. Набор конкретных рецептов не обладает универсальностью, так как в нем не используется тот факт, что разные виды пудингов могут состоять из одних базовых компонентов. Он не может расширяться, поскольку любое изменение компонента потребует переработки всех зависящих от него рецептов.

Пудинг – всего лишь метафора для того, что представляет для нас реальный интерес (т. е. контрактов), но поскольку их описание не требует никаких специальных знаний, мы продолжим использовать этот пример. «Хорошее решение» заключается в следующем:

- «Определить небольшой набор „комбинаторов“.
- Определить все пудинги в терминах комбинаторов.
- Вычислить содержимое сахара на основании комбинаторов».

Комбинатор представляет собой оператор, создающий составной объект из нескольких сходных объектов. Позаимствованное из презентации дерево на рис. 13.1 показывает, какие комбинаторы могут использоваться для описания пудингов.

Примечание

Мы разделяем тревогу читателя по поводу неаппетитной природы этого примера – особенно для автора, живущего в Париже. В оправдание можно сказать только то, что презентация была рассчитана на иностранную аудиторию, которая (наряду с незнанием метрической системы) обычно обладает весьма тривиальными кулинарными вкусами. Далее будем считать, что неразборчивость в выборе десерта не предполагает неразборчивости в выборе языка и архитектурных парадигм.

Нелистовые узлы дерева представляют комбинаторы, применяемые к поддеревьям. Например, «Взять» – комбинатор, получающий два аргумента: ингредиент («Сливки» слева, «Апельсины» справа) и количество («1 пинта» и «6»). Результатом применения комбинатора, обозна-

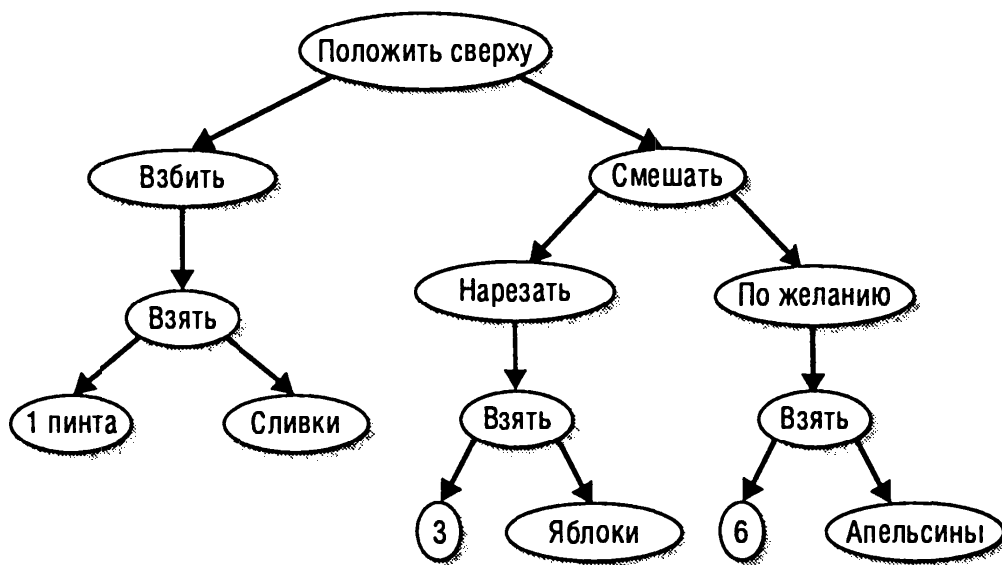


Рис. 13.1. Ингредиенты и комбинаторы, описывающие рецепт пудинга

ченного узлом дерева, является компонент пудинга (или весь пудинг), состоящий из заданного количества экземпляров компонента.

Такая структура также может быть выражена в текстовом виде, с применением «мини-языка предметной области» (DSL, Domain Specific Language) «для описания пудингов» (жирным шрифтом выделяются операторы):

"салат"	= положить_сверху украшение основная_часть"
"украшение"	= взбить (взять 1 пинта сливки)
основная_часть	= смешать яблочная_часть апельсиновая_часть
яблочная_часть	= нарезать (взять 3 яблоки)
апельсиновая_часть	= по желанию (взять 6 апельсины)"

В этой записи используется анонимная, но весьма типичная разновидность записи функционального программирования, в которой применение функции записывается аргументами функции (например, плюс a и b для применения операции плюс к a и b), а скобки используются только для группировки.

При подобном подходе такая операция, как определение содержания сахара (S), задается посредством анализа комбинаторов (по аналогии с определением математической функции для рекурсивных объектов с сохранением той же рекурсивной структуры):

"S (положить_наверх p_1 p_2)"	= $S(p_1) + S(p_2)$
S (взбить p)	= $S(p)$
S (взять q i)	= $q * S(i)$
и т. д."	

Из «и т. д.» совершенно неясно, как операторы типа S должны поступать с комбинатором по_желанию; должен существовать какой-то способ определить, содержит ли конкретная заготовка необязательный компонент, или нет. Но даже если забыть об этом обстоятельстве, описанный подход обеспечивает все преимущества, перечисленные в презентации:

- «При определении нового рецепта мы можем вычислить содержание сахара без дополнительной работы.
- Модификация S потребуется только при добавлении новых комбинаторов или новых ингредиентов».

Конечно, нашей настоящей целью являются не пудинги, а контракты. В презентации приводится общая схема, но статья содержит более подробную информацию. Она базируется на тех же идеях, примененных к более интересному набору элементов, комбинаторов и операций.

Элементами являются финансовые контракты, даты и наблюдаемые условия (например, курс обмена на определенную дату). Примеры простейших контрактов: *zero* (возможна покупка в любое время, без прав и обязательств) и *one* (c) для валюты c (немедленная выплата владельцу одной единицы c).

Примеры использования *комбинаторов* в контрактах: *or* (приобретение контракта (*or* c_1 , c_2) означает приобретение либо c_1 , либо c_2 , а срок действия контракта прекращается с истечением срока действия c_1 и c_2); *anytime* (контракт (*anytime* c) может быть приобретен в любой момент до истечения срока действия c , а его срок действия истекает одновременно с c); *truncate* (контракт (*truncate* t c) эквивалентен c за исключением того, что срок его действия истекает в более раннюю из дат t и истечения срока действия c); *get* (или (*get* c), означает приобретение c с истечением срока действия). В статье перечислено около дюжины подобных базовых комбинаторов для контрактов и другие возможные варианты для дат и условий. Это позволяет определять сложные финансовые инструменты (скажем, «европейский опцион») простым способом:

```

european t u  = get (truncate t (or u zero))

```

Операции включают дату истечения срока действия контракта и (в конечном итоге самое важная практическая польза от нашего моделирования) его оценочную серию — проиндексированную по времени последовательность ожидаемых значений стоимости контракта. Как и в случае с содержанием сахара в пудинге, функции определяются посредством анализа базовых конструкторов. Несколько примеров с применением перечисленных выше базовых элементов и комбинаторов к операции H , обозначающей функцию даты истечения срока действия («горизонт»):

<code>H (zero)</code>	$= \infty$ – специальное значение с особыми свойствами
<code>H (or c1 c2)</code>	$= \max (H (c1), H (c2))$
<code>H (anytime c)</code>	$= H (c)$
<code>H (truncate t c)</code>	$= \min (t, H (c))$
<code>H (get c)</code>	$= H (c)$

Правила вычисления оценочных серий имеют аналогичную структуру, хотя правая сторона выражений, естественно, получается более сложной и включает различные финансовые и числовые операции. За дополнительными примерами применения комбинаторов и идей функционального программирования в финансовых приложениях обращайтесь к Frankau [2008].

Оценка модульности функциональных решений

Описание из предыдущего раздела, хотя в нем опущены многие тонкости презентации и особенно статьи, может послужить основой для обсуждения архитектурных особенностей функционального подхода и их сравнения с объектно-ориентированным представлением. Далее мы будем свободно переключаться между примерами с пудингами (они позволяют немедленно понять идею) и финансовыми контрактами (типичными для реальных приложений).

Критерии расширяемости

Как указано в презентации, прямая польза такой архитектуры заключается в простоте добавления нового комбинатора: «При определении нового рецепта мы можем вычислить содержание сахара без дополнительной работы». Однако это свойство вряд ли можно назвать следствием применения функционального подхода. В нашем рассмотрении было введено понятие комбинатора, создающего пудинг или его отдельные компоненты (или контракты) из других компонентов, которые в свою очередь могут быть как атомарными, так и результатами применения комбинаторов к более элементарным компонентам.

Из статьи и презентации может показаться, что в них представлена новая идея представления финансовых контрактов, но в действительности этот подход не отличается новизной. Если перевести его в область проектирования графических интерфейсов, «плохое решение», отвергнутое в начале презентации (перечисление всех типов пудингов, отдельное вычисление содержания сахара для каждого из них и т. д.) соответствует детальной проработке каждого экрана интерактивного приложения и записи соответствующих операций – вывода, перемещения, изменения размеров, сокрытия. Никто так не поступает; любая среда проектирования графических интерфейсов предоставляет атомарные

элементы (такие как кнопки и команды меню) и операции для их рекурсивного объединения в окна, меню и другие контейнеры для формирования сложного интерфейса. По аналогии с тем, как комбинаторы в примере с пудингами определяли содержание сахара и количество калорий по ингредиентам, а комбинаторы в примере с контрактами определяли горизонт и оценочную серию сложного контракта в контексте его составляющих, операции вывода, перемещения, изменения размеров и сокрытия сложной фигуры сводятся к рекурсивному применению этих операций к компонентам. Библиотека EiffelVision (Eiffel Software: документация EiffelVision) применяет композиционные принципы особенно систематическим образом, но такой подход вряд ли можно назвать уникальным. Таким образом, статья всего лишь применяет известную методологию в новой прикладной области финансовых контактов. Однако методология не требует обязательной реализации на базе функционального программирования; подойдет любая инфраструктура с механизмом передачи управления и рекурсией.

Интересные проблемы модульности возникают не при применении существующих комбинаторов к компонентам существующих типов, а при изменении типов комбинаторов и компонентов. В презентации говорится: «Модификация S (комбинатор сахара) потребуется только при добавлении новых комбинаторов или новых ингредиентов». Вопрос в том, насколько разрушительными будут такие изменения для архитектуры.

В действительности состав значимых изменений оказывается более обширным:

- Наряду с *атомарными типами* и *комбинаторами* необходимо учитывать изменения в *операциях*: добавление функции подсчета калорий для пудингов, операций задержки для контрактов, операции поворота для графических объектов.
- Кроме добавления во всех перечисленных категориях следует также учитывать возможные операции *изменения* и *удаления*, хотя для простоты мы будем и дальше ограничиваться только операциями добавления.

Оценка функциональной методологии

Структура программ в приведенном виде достаточно проста. Она состоит из определений в форме:

$$0(a) = b_{a,0} \quad [1]$$

$$0(c(x, y, \dots)) = f_{c,0}(x, y, \dots) \quad [2]$$

для каждой операции 0 , атомарного типа a и комбинатора c . В правой части указываются соответствующие константы b и функции f . И снова для простоты атомарные типы, такие как a , будут рассматриваться как 0 -арные комбинаторы, поэтому нам достаточно рассмотреть только формулу [2]. С t основными комбинаторами (положить_наверх, взбить) и f операциями (содержание сахара, калории) потребуется $t \times f$ определений.

Независимо от выбора методологии эти $t \times f$ элементов необходимо как-то разместить. Архитектурная проблема заключается в том, как сгруппировать их по модулям для упрощения расширения и возможности повторного использования. В статье и презентации эта проблема не рассматривается. Конечно, для малых значений t и f она не критична; в этом случае все определения можно упаковать в одном модуле. При таком решении проблема расширяемости решается просто:

- Чтобы добавить комбинатор c , добавьте f определений в приведенной выше форме, по одному для каждой существующей операции.
- Чтобы добавить операцию 0 , добавьте t определений, по одному для каждого существующего комбинатора.

Такой подход плохо масштабируется, и в более крупных разработках систему придется делить на модули. В этом случае проблема расширяемости будет заключаться в выборе организации, при которой такие изменения влияют на минимальное количество модулей.

Даже при относительно малых t и f одномодульное решение препятствует повторному использованию. Если другой программе понадобится ограниченное подмножество операций и комбинаторов, она столкнется со стандартной дилеммой примитивной модуляризации:

Харибда

Копирование/вставка нужных фрагментов – но с риском того, что производные модули не будут обновлены в случае изменения оригинала (например, по такой прозаической причине, как исправление ошибки).

Сцилла

Импортирование всего модуля любыми доступными средствами включения модулей. В результате проект отягощается большим количеством «балласта», что усложняет обновления и может привести к конфликтам (допустим, в производном модуле определяется новый комбинатор или функция, а в следующей версии исходного модуля вводится конфликтное определение).

Эти наблюдения попутно напоминают нам, что универсальность тесно связана с расширяемостью. Интернет-критик функционального языка OCaml (Steingold 2007)¹ приводит конкретный пример:

Поведение модуля невозможно легко изменить за его пределами. Допустим, вы используете модуль `Time`, в котором определяется метод `Time.date_of_string` для разбора базового формата ISO8601 ("YYYYMMDD"), но хотите использовать расширенный формат ISO8601 ("YYYY-MM-DD"). Не повезло: вам придется редактировать исходную функцию – переопределить ее в своем модуле вам не удастся.

По мере роста и изменения программного продукта критическую роль начинает играть другой аспект универсальности: возможность повторного использования общих свойств. Наряду с европейскими опционами в статье вводятся «американские опционы». При описании на уровне комбинаторов они имеют другие сигнатуры (Дата → Контракт → Контракт и (Дата, Дата) → Контракт → Контракт), для каждой из них приходится определять все операции по отдельности. Однако разумно предположить, что две разновидности опционов должны иметь ряд общих свойств и операций (по аналогии с группировкой пудингов по категориям). Такие группировки упрощают моделирование и деление программы на модули, с дополнительным преимуществом (при достаточном количестве общих аспектов) в виде сокращения количества необходимых определений. Однако для этого потребуется по-новому взглянуть на предметную область задачи, чтобы кроме функций выявить важнейшие *типы*.

Такое представление будет находиться на более высоком уровне абстракции. Особенно спорной может показаться фиксация на функциях и их сигнатурах. Как сказано в статье (с сохранением курсивного выделения), «американский опцион обладает большей гибкостью, чем европейский. Обычно американский опцион предоставляет возможность приобрести контракт *в любой момент времени между двумя датами* или не приобретать его вообще». Это предполагает определение по различию: либо американские опционы являются особым случаем европейских, либо оба класса являются разновидностями более общего понятия опциона. Однако при определении в виде комбинаторов они немедленно разделяются из-за лишней даты в сигнатуре. Происходящее сродни определению концепции по реализации (в математическом, а не компьютерном смысле, но все равно с потерей абстракции и общности). Выбор типов в качестве базового механизма деления про-

¹ Цитата слегка упрощена. Включение цитаты не подразумевает подтверждения других критических замечаний на этой странице.

граммы на модули, как в объектно-ориентированных решениях, поднимает систему на более высокий уровень абстракции.

Уровни модульности

Оценка функционального программирования по критериям модульности вполне законна, потому что улучшение модульности является одним из главных аргументов в пользу этого подхода. Ранее уже приводились комментарии по этому поводу из презентации; далее приводится более общее утверждение из одной из основополагающих статей функционального программирования (Hughes, 1989), в которой сказано, что при таком подходе:

[Программы] могут делиться на модули новыми способами и, как следствие, серьезно упрощаются. В этом факте кроется ключ к силе функционального программирования – оно значительно улучшает модульность программы. Кроме того, он устанавливает цель, к которой должны стремиться функциональные программисты: уменьшение и упрощение модулей, придание им более общего характера, объединение их с использованием новых связующих элементов, которые будут описаны ниже.

«Новые связующие элементы», упоминаемые в статье Хьюза, уже встречались нам в двух описанных примерах – систематическое применение функций без состояния, включая высокоуровневые функции (комбинаторы), применяемые к другим функциям, и обширное использование списков и других рекурсивно определяемых типов, а также концепции отложенного вычисления.

При всей своей привлекательности эти методы ориентированы только на решение проблемы мелкоструктурного модульного деления. Хьюз разрабатывает функциональную версию метода Ньютона–Рафсона для вычисления квадратного корня числа N с отклонением ϵ и исходным приближением a_0 :

```
sqrt a0 eps N = within eps repeat (next N) a0)
```

с соответствующими комбинаторами `within`, `repeat` и `next` и сравнивает эту версию с написанной на Fortran программой, содержащей команды `goto`. Даже если не обращать внимания на некорректность сравнения (на момент публикации статьи язык Fortran уже считался древностью, а команды `goto` вышли из употребления), понятно, почему некоторые люди предпочитают решение, основанное на использовании малых функций, связанных комбинаторами, версии с циклами. Но другие предпочитают циклы, а поскольку речь идет о мелкой структуре программ, а не о крупномасштабном модульном делении, это скорее

вопрос стиля и вкуса, нежели фундаментальный архитектурный вопрос. В частности, проблемы демонстрации правильности в обоих случаях практически совпадают. Например, в приведенном в статье Хьюза определении получение первого элемента последовательности, отличающегося от предыдущего менее чем на `eps`:

```
within eps ([a:b:rest]) = if abs (a - b) <= eps then b
else within eps [b:rest]
```

предполагает, что расстояния между соседними элементами уменьшаются, а также совершенно точно предполагает, что одно из этих расстояний будет не больше `eps`¹. Формулировка этого свойства подразумевает наличие некоего механизма, сходного с контрактным проектированием, для связывания с функциями предусловий (тогда как в стандартных функциональных методологиях такой механизм отсутствует). Обоснование того, что гарантирует завершение в области `eps`, по сути эквивалентно обоснованию завершения соответствующего цикла в императивном решении.

В этом и предшествующих примерах ничто не относится к крупноструктурному делению на модули. В частности, лишенная состояния природа функционального программирования никак не влияет на него (ни положительно, ни отрицательно).

Преимущества функционального подхода

Из приведенных примеров видно, что у функционального подхода остаются еще четыре важных преимущества.

Первое преимущество – запись. Несомненно, привлекательность функционального программирования отчасти обусловлена компактностью определений, как в рассмотренном примере. Это сокращает количество синтаксического балласта по сравнению с объявлениями функций в типичном императивном языке. Однако это наблюдение нуждается в нескольких уточнениях:

- При рассмотрении архитектурных аспектов, как в нашем случае, запись не столь критична. Например, можно использовать функциональный стиль проектирования с императивным языком.
- Многие современные функциональные языки, такие как Haskell и OCaml, обладают сильной типизацией, вследствие чего запись неизбежно отчасти утратит компактность; например, если проекти-

¹ В цитатах из статьи Хьюза мы с согласия автора использовали современное (Haskell) обозначение списков `[a:b:rest]`, более понятное по сравнению с исходной записью вида `cons a (cons b rest)`.

ровщик не хочет полагаться на интерфейс типа (чего на стадии проектирования делать определенно не стоит), для `within` потребуются объявления типа `Double → [Double] → Double`.

- Не каждый программист будет уверенно чувствовать себя с систематической заменой многоаргументных функций функциями, возвращающими функции (в статье о финансовых контрактах это замечание проиллюстрировано сигнатурами вида $(a \rightarrow b \rightarrow c) \rightarrow \text{Obs } a \rightarrow \text{Obs } b \rightarrow \text{Obs } c$). Хотя это скорее вопрос стиля, нежели фундаментальное свойство методологии, оно встречается в этой и во многих других публикациях.

Тем не менее, лаконичная запись остается положительным свойством даже на уровне проектирования и архитектуры, и из функциональных языков программирования можно почерпнуть некоторые полезные уроки.

Второе преимущество (подчеркнутое Саймоном Пейтоном Джонсом и Диомидисом Спинеллисом в комментариях к предыдущему варианту этой главы) тоже относится к записи. Речь идет об эlegantности комбинаторных выражений определения объектов. В императивном объектно-ориентированном языке эквивалентом комбинаторного выражения вида

```
положить_сверху украшение основная_часть
```

будет инструкция

```
создать пудинг .украсить (украшение, основная_часть)
```

где процедура создания (конструктор) `украсить` инициализирует атрибуты `основа` и `украшение` заданными аргументами. Комбинатор здесь применяется скорее в описательной, нежели в императивной форме. Впрочем, на практике комбинаторная форма довольно часто применяется в объектно-ориентированном программировании в форме «методов-фабрик» (в отличие от явных команд создания объектов).

Два последних преимущества имеют более фундаментальную природу. Первое – способность манипулировать с операциями как с «полноценными сущностями», то есть как объектами программы (или как с данными). Lisp впервые показал, что это можно делать эффективно. В некоторых популярных языках была предусмотрена возможность передачи функций в качестве аргументов других функций, но она не считалась фундаментальным приемом проектирования, а иногда даже рассматривалась как пережиток самомодифицирующегося кода со всеми вытекающими подозрениями по его поводу. Современные функциональные языки демонстрируют мощь интерпретации высокоуровневых функциональных конструкций как обычных объектов программы

и развивают соответствующие системы типов. Эта часть функционального программирования оказывает самое непосредственное влияние на разработку популярных методологий программирования; как будет показано ниже, концепция агента, созданная на основе концепций функционального программирования, становится полезным добавлением в исходную объектно-ориентированную архитектуру.

Четвертым серьезным преимуществом функционального программирования являются отложенные (или «ленивые») вычисления: возможность описания потенциально бесконечных вычислений (хотя следует понимать, что любое конкретное выполнение этих вычислений будет конечным). В частности, оно предполагается в приведенном ранее определении `within`; это еще более очевидно в следующем определении `repeat`:

```
repeat f a = [a : repeat f (f a)]
```

Данная запись генерирует (в традиционной синтаксисе вызова функций) бесконечную последовательность $a, f(a), f(f(a)), \dots$. При определении `Next N x` в виде $(x + N / x) / 2$, определение `within` остановит вычисление этой последовательности после конечного количества элементов.

Идея весьма элегантная. По поводу ее обобщенного применения в программных архитектурах необходимо сделать пару замечаний.

Во-первых, следует учитывать проблему корректности. Простота написания потенциально бесконечных программ скрывает сложности, связанные с их обязательным завершением. Мы уже видели, что `within` предполагает наличие предусловия, но это предусловие, требующее, чтобы элементы стали ниже `eps`, не может иметь гарантированного конечного вычисления на бесконечной последовательности (свойство полуразрешимости). Проектировщикам приходится прибегать к всевозможным ухищрениям, о чем свидетельствует старая задача: сколько функциональных программистов потребуется, чтобы заменить лампочку? Если помните, заранее предсказать невозможно. Если еще остались функциональные программисты, предложить одному из них заменить лампочку. Если у него не получится, попробовать другого.

Во-вторых, отложенные манипуляции с бесконечными структурами возможны в нефункциональной среде проектирования без какой-либо специальной поддержки. Подход, основанный на абстрактных типах данных (называемый также объектно-ориентированным проектированием), обеспечивает возможность такого решения. Конечные последовательности и списки в библиотеках Eiffel доступны через API, основанный на понятии «курсора» (рис. 13.2).

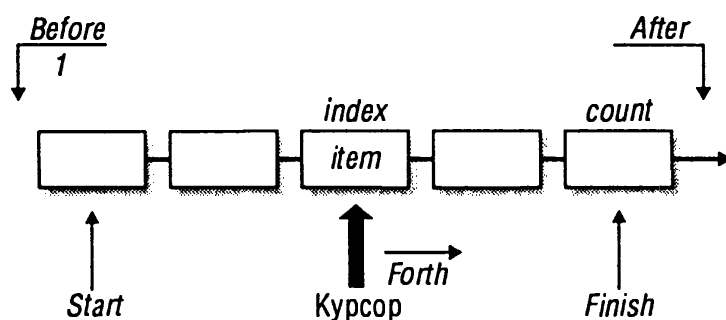


Рис. 13.2. Курсоры в списках Eiffel

Перемещение курсора осуществляется командами `start` (перемещение к первому элементу), `forth` (перемещение к следующему элементу) и `finish`. Логические проверки `before` и `after` сообщают соответственно, находится ли курсор перед первым элементом или за последним. Если ни одно условие не выполняется, то `item` возвращает элемент в текущей позиции, а `index` — его индекс.

Данная спецификация легко адаптируется для бесконечных последовательностей; для этого достаточно удалить `finish` и `after` (а также `count`, количество элементов). Так выглядит спецификация отложенного (абстрактного) класса `COUNTABLE` в библиотеке Eiffel. К числу его потомков относятся классы `PRIMES`, `FIBONACCI`, `RANDOM`; каждый предоставляет собственную реализацию `start`, `forth` и `item` (а в последнем случае возможность задать значение для раскрутки генератора псевдослучайных чисел). Для получения последовательных элементов одной из этих бесконечных последовательностей достаточно применить `start`, а затем запрашивать `item` после конечного числа применений `forth`.

Этот стиль может использоваться для моделирования любой бесконечной последовательной структуры, требующей конечного вычисления. Хотя это не исчерпывает всех возможных применений отложенного вычисления, преимущество заключается в явном определении бесконечной структуры, упрощающем обеспечение корректности отложенного вычисления.

Проблема состояния

Функциональный подход стремится к прямой зависимости от свойств математических функций, для чего он отвергает предположение, неявно присутствующее в императивных методологиях, что вычислительные операции наряду с выдачей результата (как у математической функции) могут изменять состояние вычисления — глобальное состояние, или, при более модульном подходе, некоторую его часть (например, содержимое конкретного объекта).

Хотя это свойство бросается в глаза во всех описаниях функционального программирования, оно остается скрытым в рассмотренных нами примерах – вероятно потому, что они следуют из исходного анализа задачи, который уже исключил состояние в пользу функциональных конструкций. Например, нефункциональная модель оценки стоимости финансового контракта могла бы использовать операцию, преобразующую текущее состояние с обновлением значения вместо функции, строящей последовательность.

Тем не менее мы можем сделать некоторые общие замечания по поводу этого фундаментального решения функциональных подходов. Трудно представить себе какую-либо систему, компьютеризированную или нет, лишенную понятия состояния. Можно даже сказать, что состояние является центральным понятием любых вычислений (приходилось слышать мнение [Peyton Jones, 2007], что программирование без состояния помогает решать проблемы параллелизма, но для общих выводов еще недостаточно фактов). Реальный мир не создает собственных копий в результате любого значимого события. Память наших компьютеров этого также не делает; она просто перезаписывает содержимое своих ячеек. Такие изменения состояния всегда можно смоделировать последовательностью значений, но подобное решение может выглядеть весьма неестественно (как в альтернативном ответе на приведенную ранее загадку: функциональные программисты не заменяют лампочки, а покупают новую розетку, новую электропроводку и новую лампочку).

Современные функциональные языки (в особенности Haskell) признают невозможность игнорирования состояния при таких операциях, как ввод и вывод, а также неуклюжесть более ранних попыток (Peyton Jones and Wadler, 1993). В них появилась концепция *монады* (Wadler, 1995), внедряющей исходные функции в функции более высокого порядка с более сложными сигнатурами; дополнительные компоненты сигнатур могут использоваться для хранения информации состояния, а также любых дополнительных элементов, таких как состояние ошибки (для моделирования обработки исключений) или результатов ввода/вывода.

Использование монад для интеграции состояния обусловлено той же общей идеей (хотя и использованной в противоположном направлении), что и описанная выше методология реализации отложенного поведения посредством моделирования бесконечных последовательностей абстрактными типами данных: чтобы эмулировать в инфраструктуре А возможность Т, *неявно* присутствующую в инфраструктуре В, запрограммируйте в А *явную* версию Т или ключевого механизма, делающего Т возможным. В первом случае Т соответствует бесконечным

спискам (а «ключевой механизм» – конечному вычислению бесконечных списков), а во втором – состоянию.

Концепция монады элегантна и, очевидно, полезна для семантических описаний языков программирования (и, особенно, в денотационном семантическом стиле). Однако возникает вопрос, насколько хорошо подходит это решение в качестве механизма, непосредственно используемого программистами. Здесь необходимо очень внимательно проанализировать аргументы. Очевидное возражение против монад – будто рядовых программистов трудно обучить пользоваться ими – несущественно; новаторские идеи, считавшиеся сложными на момент их появления, достаточно легко вливаются в массовое применение, когда преподаватели разрабатывают правильные способы их изложения (и рекурсия, и объектно-ориентированное программирование тоже когда-то считались недоступными для простого «программиста Джо»). Важнее понять, стоит ли игра свеч. Предоставить функциональным программистам доступ к состоянию через монады – все равно, что убедить вашу паству в необходимости целомудрия, а потом сказать, что иметь детей вообще-то неплохо, если они от вас.

Так ли уж необходимо исключать состояние? Пары наблюдений достаточно, чтобы усомниться в этом:

- *Элементарные* операции изменения состояния, такие как присваивание простых значений, имеют четкую математическую модель (логика Хоара, основанная на замене). Тем самым снижается ценность основного преимущества, обычно связываемого с программированием без состояния: упрощение математического обоснования программ.
- Для более *сложных* аспектов установления корректности архитектуры или реализации преимущества функционального подхода не столь очевидны. Например, чтобы доказать, что рекурсивное определение обладает определенными свойствами и имеет завершение, необходим эквивалент инварианта и варианта цикла. Также маловероятно, чтобы эффективные функциональные программы могли позволить себе отказаться от связанных структур данных при всех вытекающих проблемах, нетривиальных независимо от нижележащей модели программирования.

Если функциональное программирование не может значительно упростить задачу установления корректности, остается важный практический аргумент: отсутствие побочных эффектов. Это аналог понятия подстановочности равенства: в математике $f(a)$ всегда означает одно и то же для заданных f и a . Данное свойство истинно для чисто функциональных методологий. В языке программирования, в котором функции

могут иметь побочные эффекты, $f(a)$ может возвращать разные результаты при разных вызовах. Устранение такой возможности упрощает понимание текста программы, поскольку мы можем использовать логические стереотипы из математики; например, все привыкли к тому, что $g + g$ и $2 \times g$ означают одно и то же, но если функция g имеет побочные эффекты, эквивалентность уже не гарантируется. Сложности возникают не столько для средств автоматизированной проверки (которые могут обнаружить наличие у функции побочных эффектов), сколько у человека, читающего код.

Отсутствие побочных эффектов в выражениях – весьма желательная цель. Однако она не оправдывает исключение понятия состояния из вычислительной модели. Важно вспомнить правило, определенное в методе Eiffel: *принцип разделения команд и запросов* (Meyer, 1997). В этой методологии операции класса четко делятся на две группы: команды, способные изменять целевые объекты, а, следовательно, и состояние, и запросы, выдающие информацию об объекте. Команды не возвращают результата; запросы не могут изменять состояния – иначе говоря, они удовлетворяют правилу отсутствия побочных эффектов. В примере с курсором командами являются `start`, `forth`, и (в конечном случае) `finish`; запросами – `item`, `index`, `count`, `before` и (в конечном случае) `after`. Правило исключает более чем распространенную схему вызова функции для получения результата одновременно с модификацией состояния, которая, вероятно, и является основным источником проблем в императивном программировании. Сначала вы запрашиваете изменение состояния при помощи команды, а затем получаете информацию при помощи запроса (свободного от побочных эффектов). Принцип также можно сформулировать так: «*Заданный вопрос не должен изменять ответа*». Например, из него следует, что типичная операция ввода должна выглядеть так:

```
io.read_character  
Result:= io.last_character
```

Здесь `read_character` – команда, читающая символ из входного потока, а `last_character` – запрос, возвращающий последний прочитанный символ (обе функции входят в базовую библиотеку ввода/вывода). Непрерывная последовательность вызовов `last_character` гарантированно будет возвращать один и тот же результат. По теоретическим и практическим соображениям, подробно изложенным в другой работе (Meyer, 1997), принцип разделения команд и запросов является методологическим правилом, а не особенностью языка, однако он тщательно соблюдался во всех серьезных программных проектах, написанных на Eiffel, и это принесло значительные преимущества. К сожалению, этот принцип не поддерживается другими школами объектно-ориентированного

программирования (для внесения изменений в них используются вызовы функций в стиле C вместо вызовов процедур), хотя в нашем представлении он является ключевым элементом объектно-ориентированного подхода. На наш взгляд, это реальный способ обеспечения присущего функциональному программированию стремления к отсутствию побочных эффектов – так как выражения, в которых задействованы только запросы, не изменяют состояния, а, следовательно, могут пониматься как в традиционной математике или функциональном языке. При этом понятие команды признает фундаментальную роль концепции состояния для моделирования систем и вычислений.

Объектно-ориентированное представление

А теперь попробуем разобраться, как построить объектно-ориентированную архитектуру для проблем, обсуждавшихся в презентации и статье.

Комбинаторы – хорошо, а типы лучше

До настоящего момента мы имели дело с операциями и комбинаторами. Операции остаются; ключевым шагом должен стать отказ от комбинаторов и замена их типами (или классами – различия проявляются только в области параметризации). Такая замена приводит к значительному повышению уровня абстракции:

- Комбинатор описывает конкретный способ построения нового механизма из уже существующих. Комбинация определяется жестко: скажем, комбинация взять 3 яблоки из приводившегося примера связывает один элемент количества с одним элементом еды. Как упоминалось ранее, это математический эквивалент определения структуры ее точной реализацией.
- Класс определяет тип объектов перечислением его возможностей (операций). Он предоставляет абстракцию в смысле абстрактных типов данных: окружающий мир воспринимает объекты исключительно по применяемым к ним операциям, а не по тому, как они конструируются. В соответствии с принципами абстракции данных и объектно-ориентированного проектирования этот подход означает, что объекты известны не по тому, чем они *являются*, а по тому, чем они *обладают* (открытые аспекты и ассоциированные контракты). Данный подход также открывает возможность создания таксономий типов, или *наследования*, для ограничения сложности модели и эффективного использования общности между объектами.

Переходя от первого подхода ко второму, мы ничего не теряем, так как комбинаторы тривиально включаются в классы как особый случай. Достаточно определить возможности с заданием компонентов и соответствующую процедуру создания (конструктор) для объектов. Пример для take:

```
class REPETITION create
  make
  feature
    base: FOOD
    quantity: REAL
    make (b: FOOD; q: REAL)
      -- Производство элемента из quantity единиц base.
      ensure
        base = b
        quantity = q
      end
    ... Прочие возможности ...
  end
```

Для получения объекта этого типа может использоваться запись `create apple_salad.make (6.0, apple)`, эквивалентная выражению с комбинатором.

Программные контракты и параметризация

Так как наше внимание прежде всего направлено на архитектуру, эффект `make` был выражен в форме постусловия, но в действительности с таким же успехом можно было использовать секцию реализации (`do base := b ; quantity := q`). Одним из следствий правильно понимаемого объектно-ориентированного проектирования является сокращение дистанции между спецификацией и реализацией. Мы свободно используем команды, изменяющие состояние, но при этом сохраняем большую часть положительных сторон архитектуры.

В отличие от комбинатора, класс не ограничивается указанными возможностями. Например, он может включать другие процедуры создания – скажем, объединение двух повторений одного компонента:

```
make (r1, r2: REPETITION)
  -- Производство элемента объединением r1 с r2.
  require
    r1.base = r2.base
  ensure
    base = r1.base
    quantity = q
```

Предусловие означает, что смешиваемые компоненты должны относиться к одному базовому типу. Требование также можно преобразовать к статическому виду при помощи системы типов; параметризация (также поддерживаемая в типизованных функциональных языках под странным, хотя и впечатляющим названием «параметрического полиморфизма») приводит нас к следующему определению класса:

```
class REPETITION [FOOD] create
    ... См. выше ...
feature
    make (r1, r2: REPETITION [FOOD])
        ... Предусловие не требуется ...
        ... Остальное не изменилось...
end
```

Классы не только могут иметь разные процедуры создания, но обычно обладают гораздо бóльшим количеством методов. А именно, *операции* из предыдущих версий становятся методами соответствующих классов. Классы пудингов (включая классы, описывающие разновидности компонентов, такие как REPETITION) обладают такими методами, как содержание сахара (sugar) и содержание калорий (calorie_content); классы контрактов имеют такие методы, как горизонт (horizon) и стоимость (value). Необходимо сделать пару замечаний:

- Так как мы начинали с чисто функциональной модели, все методы, упоминавшиеся до настоящего момента, являются либо процедурами создания, либо запросами. Хотя функциональный стиль можно сохранить и в объектно-ориентированной инфраструктуре, в ходе разработки в нее могут быть добавлены команды – например, для изменения контракта в ответ на определенное событие (скажем, перезаключение контракта). Впрочем, этот вопрос – быть или не быть состоянию? – в целом не имеет отношения к теме модульного деления.
- В оригинале функция value порождала бесконечную последовательность. Для сохранения этой сигнатуры можно использовать результат типа COUNTABLE, эквивалент отложенного вычисления; а можно передавать value целочисленный аргумент, чтобы вызов value (i) возвращал i-е значение последовательности.

Политика деления на модули

Модульность, достигнутая до настоящего момента, демонстрирует одну из основополагающих идей объектных технологий (по крайней мере, мы ее считаем основополагающей): *объединение концепций типа и модуля* (Meyer, 1997). В своем простейшем выражении объектно-ориентированный анализ, проектирование и реализация означают, что каждый

модуль системы базируется на некотором типе или объекте, с которым работает система. Эта дисциплина устанавливает более жесткие ограничения, чем модульные средства других методологий: модуль перестает быть простой совокупностью программных элементов – операций, типов, переменных, – которые проектировщик решил держать вместе на основании некоего подходящего критерия. Модуль становится набором свойств и операций, применимых к экземплярам типа.

Класс является результатом этого слияния типов с модулями. В объектно-ориентированных языках, таких как Smalltalk, Eiffel и C# (но не C++ или Java), слияние имеет двустороннюю природу: класс не только определяет тип (или шаблон типа, если задействована параметризация), но и любой тип (включая базовые типы, скажем, целые числа) формально определяется как класс.

Классы также можно ограничить ролью типов, отдельно от модульной структуры. Это особенно справедливо для таких функциональных языков, как OCaml, предоставляющих как традиционную модульную структуру, так и механизм типов, позаимствованный из объектно-ориентированного программирования (а также Haskell с его более ограниченной концепцией класса). И наоборот, возможно снять требование, в соответствии с которым все типы определяются классами, как в C++ и Java, где базовые типы (скажем, целые числа) классами не являются. В нашем представлении объектной технологии происходит полное слияние; при этом необходимо понимать, что высокоуровневая группировка классов (пакеты Java или .NET, кластеры Eiffel) может быть необходимой, но она остается всего лишь организационным средством, а не фундаментальной конструкцией.

Такой подход подразумевает *приоритет типов перед функциями* в том, что касается определения программной архитектуры. Критериями модульного деления являются типы, в которых каждая операция (функция) присоединяется к классу, а не наоборот. Впрочем, функции добиваются компенсации через применение принципов абстрактных типов данных: класс определяется и воспринимается окружающим миром через абстрактный интерфейс (API), перечисляющий возможные операции и их формальные семантические свойства (контракты: предусловия, постусловия и, для класса в целом, – инварианты).

Логическое обоснование такой политики деления на модули заключается в том, что она улучшает модульность системы, в том числе расширяемость, возможность повторного использования и (за счет использования контрактов) надежность. Тем не менее все эти перспективы желательно проанализировать на конкретных примерах.

Наследование

Важнейшим вкладом объектно-ориентированных методов в цели модульности является механизм наследования. Предполагается, что читатель уже с ним знаком, поэтому мы напомним лишь некоторые базовые идеи и очертим их возможные применения в примерах.

Наследование организует классы в таксономии, приблизительно представляющие отношения типа «является частным случаем» – в отличие от другого базового вида отношений между классами – *клиентских* отношений, то есть использования класса через его API (операции, сигнатуры, контракты). В общем случае наследование не обязано соблюдать принцип сокрытия информации, поскольку он несовместим с представлением «является частным случаем». Некоторые авторы ограничивают применение наследования «чистой» субтипизацией, но в действительности нет ничего плохого в его применении для поддержки стандартного механизма включения модулей. В Eiffel используется механизм «неполного наследования» (Eema International, 2006), который запрещает полиморфизм, но сохраняет все остальные свойства наследования. Двойственная роль наследования вполне соответствует двойственной роли классов как типов и модулей.

В обоих качествах наследование служит для отражения общих качеств. Скажем, для описания элементов нашей вымышленной таксономии пудингов можно использовать граф наследования, показанный на рис. 13.3.

Обратите внимание на распределение ролей между наследованием и клиентскими отношениями. Фруктовый салат (FRUIT_SALAD) является

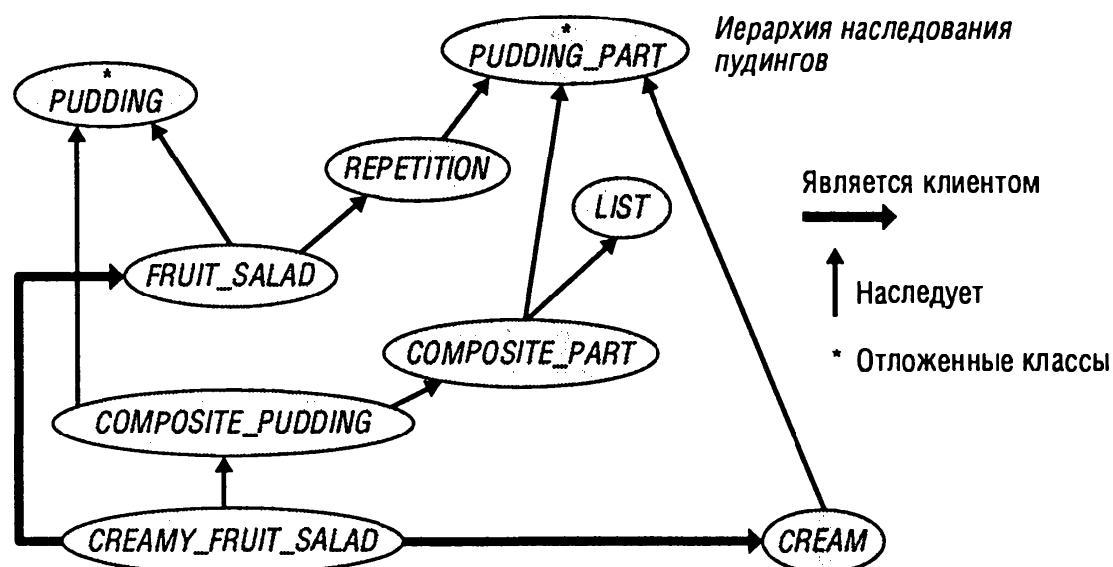


Рис. 13.3. Диаграмма классов ингредиентов пудинга

пудингом и повторением (REPETITION) из более раннего примера. Повторение является частным случаем не пудинга (PUDDING), а части пудинга (PUDDING_PART). Некоторые, но не все части пудинга (скажем, составной пудинг COMPOSITE_PUDDING) также являются пудингами. Фруктовый салат является как пудингом, так и повторением (фруктовых частей). С другой стороны, фруктовый салат со сливками (CREAMY_FRUIT_SALAD) не является фруктовым салатом, если понимать под этим термином пудинг, состоящий из одних фруктов. В нем есть и фруктовый салат, и сливки, что изображено на рисунке соответствующими клиентскими связями. Он является составным пудингом, так как это понятие обозначает изделия, которые состоят из нескольких частей (как и более общее понятие COMPOSITE_PART) и при этом одновременно являются пудингами. В данном случае частями, как показывают клиентские связи, являются фруктовый салат и сливки.

Аналогичный подход может быть применен к примеру с контрактами. Типы контрактов делятся на категории: «бескупонные облигации», «опционы» и т. д., полученные на основании тщательного анализа экспертов в этой предметной области.

Множественное наследование абсолютно необходимо для такой объектно-ориентированной формы моделирования. Обратите особое внимание на определение COMPOSITE_PART, в котором применяется стандартный паттерн для описания подобных составных структур (см. Meyer, 1997, 5.1):

```
class COMPOSITE_PART inherit
    PUDDING_PART
    LIST[PUDDING_PART]
feature
    ...
end
```

В квадратных скобках задаются обобщенные параметры. Составная часть одновременно является как частью пудинга со всеми ее свойствами и операциями (содержание сахара и т. д.), так и списком частей пудинга, также со всеми операциями, характерными для списков: перемещениями курсора (start и forth), запросами (item и index), командами вставки и удаления элементов. Элементы списка могут быть пудингами любых из существующих видов, включая, рекурсивно, составные части. Это позволяет применять методы полиморфизма и динамической привязки, которые будут рассмотрены ниже. Обратите внимание на полезность совмещения параметризации и наследования. Также множественное наследование должно быть представлено полноценным механизмом классов, а не ограниченной интерфейсной формой (как в Java и .NET), которая для данного случая не подходит.

Полиморфизм, полиморфные контейнеры и динамическая привязка

Вклад наследования и параметризации в расширяемость частично представлен методами полиморфизма и динамической привязки, как показывает следующая версия `sugar_content` класса `COMPOSITE_PART` (рис. 13.4):

```
sugar_content: REAL
do
  from start until after loop
    Result := Result + item.sugar_content
  forth
end
end
```

Операции класса `LIST` применяются непосредственно к производному от него классу `COMPOSITE_PART`. Результат `item` может относиться к любому из типов, производных от `PUDDING`; так как в дальнейшем переменная может использоваться для обозначения объектов разных типов, она называется полиморфной переменной (в данном случае точнее назвать ее *полиморфным запросом*). Вся структура `COMPOSITE_PART`, содержащая элементы разных типов, называется *полиморфным контейнером*. Полиморфные контейнеры возможны благодаря комбинации полиморфизма, который сам по себе является результатом наследования и параметризации (так как речь идет о двух совершенно разных механизмах, обозначение параметризации принятым в функциональном программировании термином «параметрический полиморфизм» приведет к путанице).

Полиморфизм `item` подразумевает, что последовательные вызовы `item.sugar_content` будут применяться к объектам разных типов; в соответствующих классах могут быть определены разные версии запроса `sugar_content`. *Динамическая привязка* гарантирует, что такие вызовы будут в каждом случае применены к правильной версии в зависимости

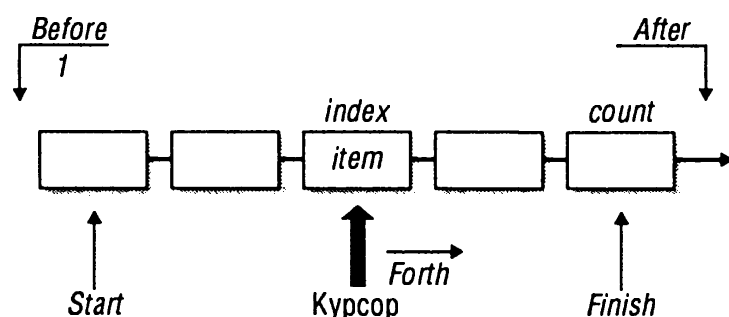


Рис. 13.4. Полиморфный список с курсорами

от фактического типа объекта. Если часть сама по себе является составной, это будет приведенная выше версия, примененная рекурсивно, но также может быть и любая другая – например, версия для CREAM.

Как и в большинстве современных подходов к объектно-ориентированному проектированию, полиморфизм работает под управлением системы типов. Значения элемента относятся к переменному типу, но только среди потомков PUDDING, как указывает параметр COMPOSITE_PART. Этот аспект разработки повлиял как на мир функционального программирования, так и на объектно-ориентированный мир: применение все более усложняющихся систем типов (по-прежнему основанных на небольшом числе простых концепций, таких как наследование, полиморфизм и параметризация) для отражения все большего объема сведений об архитектуре системы в ее структуре типов.

Отложенные классы и методы

На приведенной ранее структурной диаграмме классы PUDDING и PUDDING_PART помечены как «отложенные» (deferred) (звездочка в синтаксисе объектно-ориентированного моделирования BON [Walden and Nerson, 1994]). Это означает, что они не имеют полностью определенной реализации; также используется термин «абстрактный класс». Отложенный класс обычно обладает отложенными методами, т. е. у него имеется сигнатура и (что еще важнее) контракт, но нет реализации. Реализации появляются в неотложенных («конкретных») классах-потомках, адаптированные к тем решениям, которые принимаются каждым конкретным классом для реализации общих концепций, определяемых абстрактным классом. В нашем примере оба класса PUDDING и PUDDING_PART обладают отложенными методами sugar_content и calories; потомки конкретизируют их – например, в COMPOSITE_PART содержание сахара определяется суммированием содержаний всех частей, как показано выше. В COMPOSITE_PUDDING, наследующем эту версию от COMPOSITE_PART и отложенную версию от PUDDING, реализация определяется конкретной версией, обладающей более высоким приоритетом.

Примечание

При наследовании двух методов с одинаковыми именами возникает конфликт имен, который должен быть разрешен посредством переименования. Исключением является ситуация, в которой один из методов является отложенным, а другой – конкретным. В этом случае формируется один метод с имеющейся реализацией. Именно в таких применениях механизма наследования перегрузка имен создает массу хронических сложностей, и возникает впечатление, что этот механизм не должен присутствовать в объектно-ориентированных языках.

Отложенные классы сложнее концепции «интерфейсов» Java и .NET. Во-первых, они могут связываться с контрактами, ограничивающими будущую конкретизацию, а, во-вторых, в них могут присутствовать конкретные методы, вследствие чего отложенные классы заполняют весь спектр между полностью отложенным классом, ограничивающимся чистым описанием реализации, и конкретным классом, полностью определяющим реализацию. Возможность описания частичных реализаций исключительно важна для объектно-ориентированного проектирования и архитектуры.

В примере с финансовыми контрактами `CONTRACT` и `OPTION` будут естественными кандидатами для оформления в виде отложенных классов, хотя они и не обязаны быть *полностью* отложенными.

Оценка и улучшение модульности в объектно-ориентированных архитектурах

В предыдущем разделе представлен краткий обзор применения объектно-ориентированных архитектурных методов в наших примерах. Теперь необходимо проанализировать схематичный результат в свете критериев модульности, представленных в начале обсуждения. Надежность в основном зависит от системы типов и контрактов; наше внимание будет сосредоточено на возможностях повторного использования и расширения.

Повторное использование операций

Одно из важнейших последствий применения наследования заключается в возможности перемещения общих методов на самый верхний возможный уровень абстракции. Далее потомкам не обязательно повторять реализацию унаследованных методов: они просто наследуют их «как есть». Если им понадобится изменить реализацию с сохранением функциональности, они просто переопределяют унаследованную версию. Под «сохранением функциональности» в данном случае понимается соблюдение исходных контрактов независимо от того, конкретизирована переопределяемая версия или все еще остается отложенной. Такой подход хорошо сочетается с динамической привязкой: клиент может использовать операцию на верхнем уровне абстракции (например, `my_pudding.sugar_content` или `my_contract.value`), не зная, какая версия операции используется, в каком классе и была ли она определена в этом классе или унаследована им.

Благодаря наследованию общих аспектов количество определений может оказаться существенно ниже максимального значения $t \times f$. Любое

сокращение, безусловно, полезно. Общее правило проектирования программных продуктов гласит, что любые повторения всегда потенциально вредны, поскольку они становятся источником будущих трудностей с управлением конфигурацией, сопровождением и отладкой (если в оригинале будет допущена ошибка, ее также придется исправлять в каждой из копий). Как справедливо заметил Дэвид Парнас, копирование/вставка – враг программиста.

Степень реального сокращения очевидным образом зависит от качества структуры наследования. Здесь следует руководствоваться принципами абстрактных типов данных: поскольку ключом к определению типов при объектно-ориентированном проектировании является анализ применимых операций, в правильно спроектированной иерархии наследования на верхний уровень будут выведены классы, содержащие общие аспекты многих разновидностей.

У этой методологии не существует четкого эквивалента в функциональной модели. При использовании комбинаторов необходимо определять разновидность каждой операции для каждой комбинации с повторением всех общих операций.

Расширяемость: добавление типов

Насколько хорошо объектно-ориентированная форма архитектуры поддерживает расширяемость? Одним из самых распространенных видов расширения системы является добавление новых типов: новой разновидности пудинга, новой части пудинга, нового финансового контракта. В подобных ситуациях объектная технология проявляется во всей красе. Просто найдите в структуре наследования место, к которому новая разновидность лучше всего подходит (в смысле наличия большинства общих операций) и напишите новый класс, который наследует некоторые методы, переопределяет или изменяет те, для которых он имеет собственные разновидности, и добавляет все новые методы и инварианты, соответствующие новой логической сущности.

Динамическая привязка снова играет важную роль; преимущество объектно-ориентированного подхода заключается в том, что он избавляет клиентские классы от необходимости выполнять многоуровневое ветвление для выполнения операций: «если это фруктовый салат, то обработать его так, иначе, если это флан, то обработать его этак, иначе...». Такие проверки должны повторяться для каждой операции и, что еще хуже, – должны обновляться в каждом клиенте и в каждой операции при каждом добавлении или изменении типа. Такие структуры, требующие, чтобы клиентские классы обладали полной информацией о структуре концепций, от которых они зависят, становились главным

источником деградации и старения архитектур до появления объектно-ориентированных методологий. Динамическая привязка решает проблему; клиентское приложение запрашивает `my_contract.value`, а внутренняя реализация сама выбирает подходящую версию.

Ни одна другая методология программных архитектур не сравнится с изяществом этого решения, объединяющего самые сильные стороны объектно-ориентированных методологий.

Расширяемость: добавление операций

Оценки удобства расширения объектных архитектур отчасти (помимо таких механизмов, как сокрытие информации и параметризация, а также центральная роль контрактов) базируются на предположении о том, что самые значительные изменения в жизни системы относятся к только что рассмотренной разновидности: введению типов, которые используют некоторые операции существующих типов, но также могут определять новые операции. В самом деле, практический опыт говорит о том, что именно они являются самым частым источником нетривиальных изменений в реальных системах, в которых проявляются преимущества объектно-ориентированных методов перед другими. Но как насчет другого случая: добавления операций в существующие типы? Допустим, некоторое клиентское приложение, использующее концепцию пудинга, может захотеть вычислить стоимость изготовления различных пудингов, хотя сами классы пудингов не имеют соответствующего метода.

Функциональное программирование справляется с добавлением операций не лучше и не хуже, чем с добавлением типов: все сводится к увеличению на 1 значения f вместо t . Однако для объектно-ориентированных решений такая нейтральность не характерна. Простейшее решение заключается в добавлении новой возможности на правильно выбранном уровне иерархии. Однако при этом возможны два потенциальных недостатка:

- Так как наследование создает относительно сильную привязку («является частным случаем») между классами, изменение распространяется на всех существующих потомков. В общем случае добавление нового метода в класс, находящийся на высоком уровне иерархической структуры, может оказаться делом весьма нетривиальным.
- Такое решение попросту невозможно, если автору клиентской системы не разрешено изменять исходные классы или он не имеет доступа к их тексту. Такая ситуация часто возникает на практике, особенно если классы сгруппированы в библиотеки (скажем, биб-

лиотеку финансовых контрактов). Неразумно разрешать изменять библиотеку автору каждого приложения, в котором она используется.

Основных объектно-ориентированных методов (Meyer, 1997) здесь недостаточно. Стандартное и часто применяемое на практике объектно-ориентированное решение основано на паттерне «Посетитель» (Gamma et al., 1994). Следующая схема, хотя и не совсем точно совпадает со стандартным представлением, дает общее представление об идее (приводится по материалам книги Мейера «Touch of Class: An Introduction to Programming Well» (2008), вводного учебника по программированию для первокурсников – отсюда видно, насколько фундаментальными стали эти концепции). На рис. 13.5 изображены основные участники паттерна.

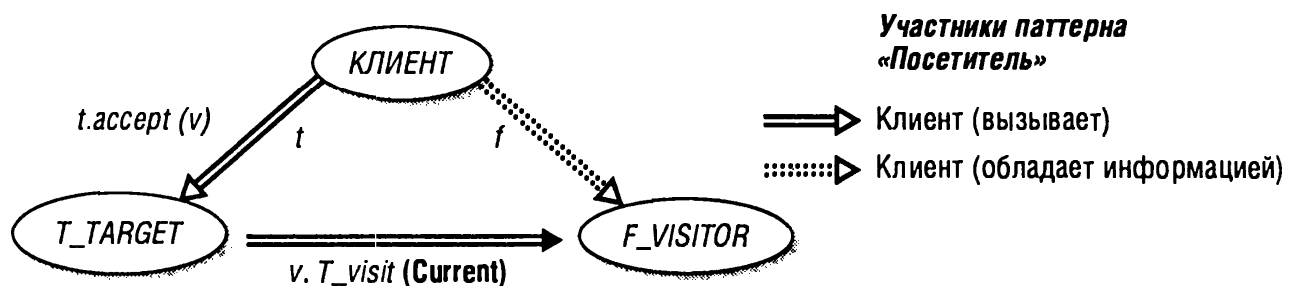


Рис. 13.5. Акторы паттерна «Посетитель»

Паттерн превращает дуэт приложения (класс CLIENT) и существующих типов (T_TARGET для конкретного типа T, которым может быть PUDDING или CONTRACT в наших примерах) в трио; для этого вводится класс-посетитель F_VISITOR для каждой новой операции F (например, COST_VISITOR). Класс приложения (CLIENT) вызывает операцию для целевого объекта, передавая посетителя в аргументе:

```
my_fruit_salad.accept(cost_visitor)
```

Команда `accept (v: VISITOR)` выполняет операцию, вызывая для своего аргумента `v` (`cost_visitor` в данном примере) функцию `FRUIT_SALAD_visit`, имя которой определяет тип целевого объекта. Функция является частью класса, описывающего целевой класс, в данном случае `FRUIT_SALAD`; она применяется к объекту соответствующего типа, который передается в аргументе `T_visit. Current` на рисунке – принятое в Eiffel обозначение текущего объекта (аналог `this` или `self` в других языках). Целевой класс вызова (`v` на рисунке) определяет операцию, используя объект соответствующего типа посетителя (например, `COST_VISITOR`).

Ключевым вопросом при оценке расширяемости в программных архитектурах всегда является распределение информации. Метод может

достигнуть расширяемости только за счет ограничения количества информации, которой модули должны располагать друг о друге (чтобы добавление или изменение модуля имело минимальные последствия для существующих структур). Чтобы понять сложную хореографию паттерна «Посетитель», полезно посмотреть, что должен или не должен знать каждый участник:

- Целевой класс знает конкретный тип, а также его контекст в иерархии типов (так как, например, `FRUIT_SALAD` наследует от `COMPOSITE_PUDDING`, а `COMPOSITE_PUDDING` — от `PUDDING`). Он *не знает* о новых операциях, вызываемых извне, таких как вычисление стоимости изготовления пудинга.
- Класс-посетитель знает все о конкретной операции (вычисление стоимости) и предоставляет ее разновидности для диапазона типов, обозначая объекты при помощи аргументов: именно здесь обнаруживаются такие функции, как `fruit_salad_cost`, `flan_cost`, `tart_cost` и т. д.
- Класс-клиент должен применять заданную операцию к объектам указанных типов, поэтому он должен знать эти типы (только факт их существования, но не другие свойства) и операции (только факт их существования и применимость к заданным типам, но не конкретные алгоритмы в каждом случае).

Некоторые из необходимых операций, такие как `accept` и `T_visit`, предоставляются предками. На рис. 13.6 изображена общая диаграмма наследования (имя класса `FRUIT_SALAD` сокращено до `SALAD`).

Такая архитектура обычно используется для включения новых операций в существующей структуре со многими вариантами наследования без необходимости изменять структуру для каждой операции. Стандартный пример применения встречается в области обработки языков (компиляторы и другие инструменты IDE), где нижележащей структурой является абстрактное синтаксическое дерево AST (Abstract Syntax Tree): необходимость обновлять класс AST каждый раз, когда новому инструменту потребуется для его собственных целей выполнить операцию перебора узлов, с применением к каждому узлу операции по желанию этого инструмента (это называется «посещением» узлов, что объясняет название паттерна «Посетитель»).

Чтобы эта схема работала, клиенты должны иметь возможность выполнить `t.accept(v)` для любого `t` любого целевого типа. Отсюда следует, что все целевые типы наследуют от общего класса (`PUDDING` в нашем примере), в котором `accept` объявляется в отложенной форме. Это довольно щекотливое требование, потому что целью всего построения было именно предотвращение модификации существующих целевых классов.

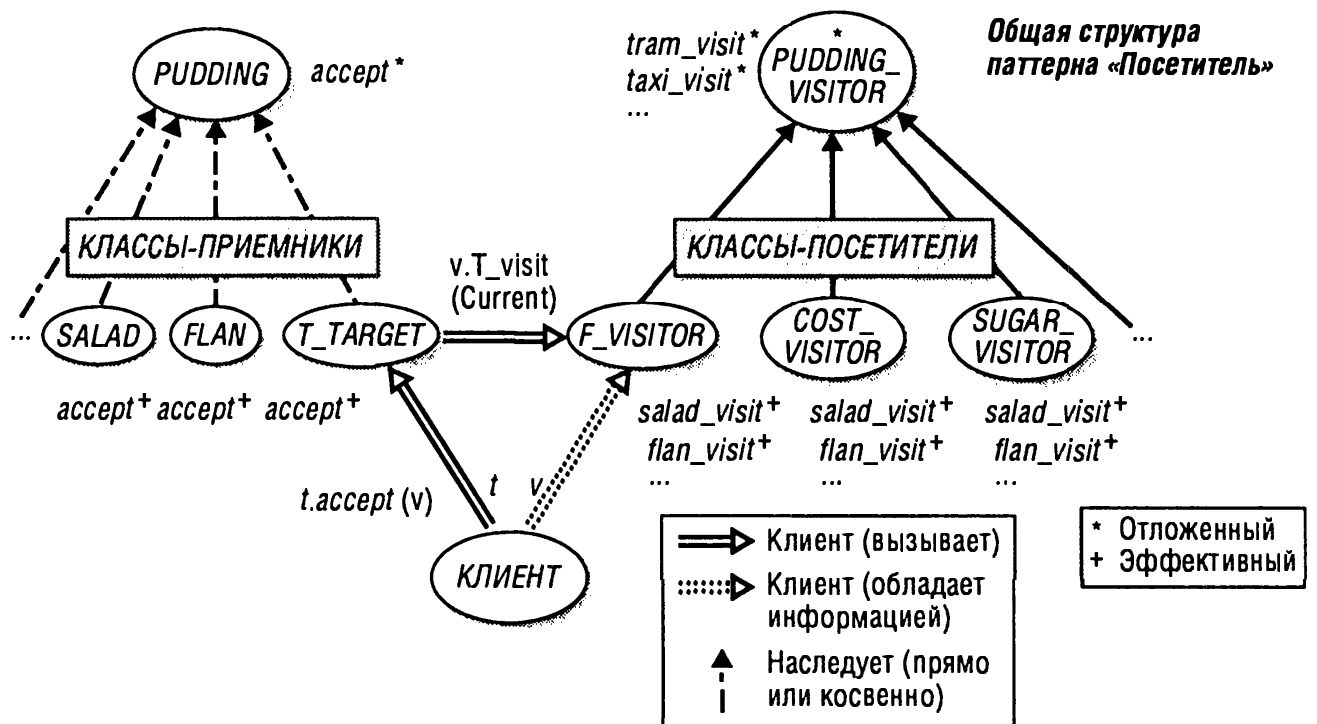


Рис. 13.6. Общая диаграмма: архитектура для конструирования пудингов

Проектировщики, использующие паттерн «Посетитель», обычно считают это требование приемлемым, поскольку для его выполнения достаточно проследить за тем, чтобы все задействованные классы имели общего предка (довольно часто это требование выполняется изначально, если они представляют собой разновидности общей концепции – такой как PUDDING или CONTRACT), и добавить в предка всего *один* отложенный метод *ассерт*.

Паттерн «Посетитель» широко применяется на практике. Пусть читатель сам судит о том, насколько он «красив». На наш взгляд, бывают и красивее. Основные претензии:

- Необходимость общего предка со специальным методом *ассерт* в классах предметной области, которые не должны отягощаться концепциями, не относящимися к области применения (будь то пудинги, финансовые контракты или что-нибудь еще).
- Еще большее беспокойство вызывает размножение классов с многочисленными миниатюрными классами F_VISITOR, воплощающими крайне специфическую информацию (специальная операция над набором специальных типов). В контексте общей программной архитектуры это воспринимается как загрязнение.

Чтобы избежать загрязнения, необходимо включить в базовую объектно-ориентированную инфраструктуру принципиально новую концепцию: агентов.

Агенты: упаковка операций в объектах

Основная идея агентов (включенных в базовую объектно-ориентированную инфраструктуру Eiffel в 1997 году; в С# используется термин «делегаты») выражается словами, напоминающими литературу по функциональному программированию: операции (функции в функциональном программировании, методы в объектно-ориентированном программировании) рассматриваются как «полноценные сущности». В контексте ОО полноценными сущностями во время выполнения являются только объекты, по своей статической структуре соответствующие классам.

Механизм агентов

Агентом называется объект, представляющий метод некоторого класса, готовый к вызову. Вызов метода $x.f(u, \dots)$ полностью определяется именем метода f , целевым объектом x и аргументами u, \dots . Агентское выражение, в котором указывается f и могут указываться целевой объект и аргументы (нуль, некоторые или все из них), называется *закрытым*. Все остальные выражения, не указанные в определении агента, называются *открытыми*. Выражение обозначает объект, представляющий метод с закрытыми аргументами, которыми присвоены заданные значения. Одной из операций, которые могут выполняться с агентом, является операция *call*, представляющая вызов f ; если у агента имеются открытые аргументы, соответствующие значения должны быть переданы в аргументах *call* (для закрытых аргументов используются значения, уже заданные в определении агента).

Простейший пример агентского выражения — *agent f*. В данном случае все аргументы являются открытыми, но целевой объект закрыт. Следовательно, если a является агентским выражением (в результате присваивания $a := \text{agent } f$ или вызова $p(\text{agent } f)$ с формальным аргументом a), то выражение $a.\text{call}([u, v])$ приводит к тому же эффекту, что и $f(u, v)$. Конечно, различие заключается в том, что $f(u, v)$ прямо задает имя метода (хотя динамическая привязка означает, что он может быть одной из разновидностей известного метода), тогда как в форме с агентами a — всего лишь имя, которое может быть получено из другого программного модуля. Таким образом, в этой точке программы о методе неизвестно ничего, кроме его сигнатуры (и, возможно, контракта). Так как *call* является библиотечной функцией общего назначения, ей должен передаваться один аргумент. Проблема решается использованием *кортежа (tuple)*, в данном случае двухэлементного кортежа $[u, v]$. В такой форме, *agent f*, целевой объект закрыт (им является текущий объект), а оба аргумента открыты.

Другая разновидность — `agent x.f`. Здесь аргументы тоже открыты, а целевой объект закрыт: на этот раз им является объект `x` вместо текущего объекта. Чтобы сделать целевой объект открытым, используется запись `agent {T}.f`, где `T` относится к типу `x`. В этом случае `call` понадобится кортеж из трех аргументов: `a.call ([x, u, v])`. Чтобы оставить некоторые аргументы открытыми, можно использовать аналогичную запись вида `agent x.f ({U}, v)` (типичный вызов имеет вид `a.call ([u])`), но, поскольку тип `U` объекта `u` очевиден из контекста, указывать его явно не обязательно; достаточно поставить вопросительный знак, как в выражении `agent x.f (?, v)`. Отсюда также следует, что исходные формы со всеми открытыми аргументами, `agent f` и `agent x.f`, являются сокращениями для `agent f (?, ?)` и `agent x.f (?, ?)`.

Механизм `call` использует динамическую привязку; используемая версия `f`, как и при «неагентском» вызове, зависит от динамического типа целевого объекта.

Если `f` представляет запрос, а не команду, для получения результата вызова `f` от соответствующего агента можно использовать `item` вместо `call`, например `a.item ([x, u, v])` (вызов с возвращением значения результата); также можно использовать `call`, а затем обратиться к `a.last_result`, которое согласно принципу разделения команд и запросов вернет то же значение без дополнительных вызовов `call` при последующих вызовах.

Возможно и нетривиальное использование агентов: вместо того чтобы определять агента на базе существующего метода `f`, можно записать его во встроенном коде, как в выражении `editor_window.set_mouse_enter_action (agent do text.highlight end)`, демонстрирующем типичное применение агентов в графических интерфейсах; это базовый стиль событийного программирования в библиотеке `EiffelVision` (Eiffel Software: документация `EiffelVision`). Механизм встроенной записи агентов аналогичен лямбда-выражениям в функциональных языках: записанные вами операции становятся непосредственно доступными для программного кода как значения, с которыми можно работать как с любыми другими «полноценными сущностями».

В более обобщенной формулировке агенты позволяют объектно-ориентированным инфраструктурам определять высокоуровневые функциональные объекты — такие же как в функциональных языках и обладающие такими же выразительными возможностями.

Область применения агентов

Агенты оказались важным и естественным компонентом базовых объектно-ориентированных механизмов. В частности, они широко используются в следующих целях:

- Итеративный перебор: применение переменной операции, которая естественным образом представляется агентом, ко всем элементам контейнерной структуры.
- Программирование GUI (см. выше).
- Математические вычисления – например, интегрирование некоторой функции, представленной агентом, по некоторому интервалу.
- Рефлексия, когда агенты предоставляют свойства методов (не только возможность вызывать их через `call` и `item`) и классов.

Агенты сыграли важную роль в нашем исследовании того, как заменить паттерны проектирования компонентами многократного использования (Arnout 2004; Arnout and Meyer, 2006; Meyer, 2004; Meyer and Arnout, 2006). Идея в том, что если проектировщику приложения понадобится паттерн, то ему придется изучать этот паттерн во всех подробностях (включая архитектуру и реализацию) и строить «с нуля» в своем приложении, тогда как компонент может просто использоваться через API. Среди успешных примеров такого рода можно назвать паттерн проектирования «Наблюдатель» (Meyer, 2004; Meyer, 2008); никому, кто видел решение на базе агентов, уже не захочется снова пользоваться этим паттерном. Другие примеры – паттерны «Фабрика» (Arnout and Meyer, 2006) и «Посетитель» – будут рассматриваться ниже.

Агентская библиотека для замены паттерна «Посетитель»

Механизм агентов предоставляет гораздо более удачное решение проблемы, которая несколько неуклюже решается паттерном «Посетитель», а именно – добавление операций к существующим типам без изменения исходных классов. Решение подробно описано у Мейера и Арну (2006) и доступно в виде библиотеки с открытым кодом на сайте Цюрихского федерального технологического института <http://se.ethz.ch>.

Итоговый клиентский интерфейс отличается исключительной простотой. В целевые классы (PUDDING, CONTRACT и т. д.) не нужно вносить никакие изменения; метод ассерт более не требуется. Классы можно использовать непосредственно в их текущем виде; больше нет размножения классов-посетителей, есть только единый библиотечный класс VISITOR всего с двумя методами для основного использования, `register` и `visit`.

Проектировщику клиента не нужно разбираться во внутреннем устройстве этого класса или беспокоиться о реализации паттерна «Посетитель». Достаточно применить базовую схему использования API:

1. Объявить переменную, представляющую объект-посетитель, указать высокоуровневый тип цели при помощи параметра VISITOR и создать соответствующий объект:

```
pudding_visitor: VISITOR [PUDDING]
create pudding_visitor
```

2. Для каждой операции, выполняемой с объектами определенного типа в целевой структуре, зарегистрировать соответствующего агента у посетителя:

```
pudding_visitor.register (agent fruit_salad_cost)
```

3. Чтобы выполнить операцию с определенным объектом (обычно в процессе перебора), используйте метод visit библиотечного класса VISITOR:

```
pudding_visitor.visit (my_pudding)
```

Вот и весь интерфейс: один объект-посетитель, регистрация применимых операций и единственная операция visit. Эта простота объясняется тремя свойствами.

- Запись применяемых операций (таких как fruit_salad_cost) не должна зависеть от выбора архитектуры. Часто они будут доступны в виде функций, что сделает возможной запись agent fruit_salad_cost; если нет (особенно если это очень простые операции), клиент может воспользоваться встроенным кодом агента. В любом случае необходимость в лишних методах T_visit отпадает.
- На первый взгляд немного странно, что для добавления посетителя достаточно всего одного класса VISITOR с одной функцией register. В решении, основанном на паттерне «Посетитель», вызов t.accept(v) и t определяли тип цели (конкретный вид пудинга), но register такую информацию не указывает. Как механизм находит правильную разновидность применяемой операции (стоимость фруктового салата или стоимость флана)? Ответ на этот вопрос является следствием рефлексивных свойств механизма агентов: объект агента воплощает всю информацию об ассоциированном методе, включая его сигнатуру. Таким образом, agent fruit_salad_cost включает информацию о том, что эта функция применима к фруктовым салатам (из сигнатуры fruit_salad_cost (fs: FRUIT_SALAD), также доступной в случае встроенного агента из его текста). Это позволяет организовать внутренние структуры данных VISITOR таким образом, что при вызо-

ве вида `pudding_visitor.visit(my_pudding)` функция `visit` найдет правильную функцию (или функции) на основании динамического типа цели, в данном случае `pudding_visitor: VISITOR[P]` для конкретного типа пудинга `P`, также соответствующую типу объекта, динамически связанному с аргументом, в данном случае полиморфным `my_pudding` (за этим на статическом уровне проследит система типов).

- Методология также пользуется преимуществами наследования и динамической привязки: если функция зарегистрирована для обобщенного типа пудинга (допустим, `COMPOSITE_PUDDING`) и нет функции, зарегистрированной для более конкретного типа (например, если стоимость всех составных пудингов может вычисляться одним способом), `visit` использует самое близкое совпадение.

В том виде, в котором механизм описан, он дополняет традиционный объектно-ориентированный механизм. Когда проблема заключается в добавлении типов, предоставляющих разновидности существующих операций, наследование и динамическая привязка творят чудеса. Для проблемы добавления операций в существующие типы без изменения этих типов подойдет описанное решение.

Применяя упоминавшийся выше критерий оценки модульности по распределению информации (кто и что должен знать?), мы видим, что в этом подходе:

- Целевые классы знают только о фундаментальных операциях (таких как `sugar_content`), характеризующих соответствующие типы.
- Приложению необходимо знать только интерфейс используемых им целевых классов, а также два важнейших метода, `register` и `visit`, библиотечного класса `VISITOR`. Если ему понадобятся новые операции с целевыми типами, не учтенные при проектировании целевых классов (такие как `cost` из нашего примера), ему необходимо только предоставить разновидности операций для целевых типов, представляющих интерес. При этом необходимо понимать, что в отсутствие переопределяющей регистрации для конкретных типов будут использоваться обобщенные операции.
- Библиотечный класс `VISITOR` ничего не знает ни о типах целей, ни о приложениях.

Кажется, сократить необходимую информацию для разных частей системы уже невозможно. По нашему мнению, остается всего один открытый вопрос: должен ли столь фундаментальный механизм оставаться доступным на уровне библиотеки или для него следует определить специальную языковую конструкцию?

Оценка

При первом появлении агентов возникли опасения, что они могут создать избыточность, а, следовательно, и путаницу, предлагая альтернативные решения в ситуациях, решаемых стандартными средствами ОО (эти опасения особенно сильны в Eiffel, проектирование которого велось по принципу предоставления «одного хорошего способа для решения любой задачи»). Этого не произошло: агенты заняли свое законное место в объектно-ориентированном арсенале, а проектировщики без особого труда определяют, когда они уместны, а когда нет.

На практике любые нетривиальные применения агентов (в частности, уже упоминавшиеся замены паттернов) также зависят от параметризации, наследования, полиморфизма, динамической привязки и других нетривиальных объектно-ориентированных механизмов. Все это укрепляет наше убеждение в том, что данный механизм является необходимым компонентом успешных объектных технологий.

Примечание

Противоположное мнение приводится в технической статье Sun, объясняющей, почему в Java не нужны аналоги агентов или делегатов (Sun Microsystems, 1997). В статье показано, как этот механизм эмулируется при помощи «внутренних классов» Java. Статья интересная и хорошо аргументированная, но, на наш взгляд, ей в основном удается доказать прямо противоположный тезис. Внутренние классы действительно справляются с этой задачей, но, как и при устранении паттерна «Посетитель» с его размножением крошечных классов, приведенное решение с использованием внутренних классов наглядно демонстрирует простоту, элегантность и модульность, присущие решению с использованием агентов.

Агенты, как упоминалось ранее, позволяют объектно-ориентированным инфраструктурам предоставлять те же выразительные возможности, как в функциональном программировании, при помощи обобщенного механизма определения высокоуровневых функциональных объектов (операций, которые могут использовать операции, рекурсивно проявляющие то же свойство, в качестве входных и выходных данных). Даже у лямбда-выражений имеются свои аналоги в виде встроенных агентов. Эти механизмы формировались под очевидным влиянием функционального программирования и теоретически должны вызвать энтузиазм у его сторонников, хотя существуют опасения, что некоторые разработчики будут считать их данью уважения, которую порок платит добродетели (La Rochefoucauld, 1665). Если не обращать внимания на синтаксис, единственное принципиальное различие заключается в том, что агенты могут инкапсулировать не только чистые

функции (запросы), но и команды. Тем не менее обеспечение полной чистоты (отсутствия побочных эффектов) не имеет особого отношения к обсуждениям архитектуры, по крайней мере при соблюдении принципа разделения команд и запросов, сохраняющего главную практическую ценность – отсутствие побочных эффектов у выражений – без насильственного перевода модели с состоянием в искусственный мир моделей без состояния.

Возможно, агенты являются «последним штрихом» в картине вклада объектных технологий в модульность, но это всего лишь один из ее элементов – как кратко представленных в дискуссии, так и тех, которые в ней не упоминались. Именно комбинация этих элементов, выходящая за рамки того, что может предложить функциональный подход, делает объектно-ориентированное проектирование лучшей из имеющихся методологий для построения красивых архитектур.

Благодарности

Я благодарен всем, кто высказал свои важные замечания по поводу чернового варианта этой статьи; как известно, благодарность еще не означает одобрения (это особенно очевидно в случае гуру функционального программирования, которые любезно поделились своим конструктивным мнением, притом что мои аргументы, как я подозреваю, их несколько не поколебали). Особенно полезными были замечания Саймона Пейтона Джонса (Simon Peyton Jones), Эрика Мейера (Erik Meijer) и Диомидиса Спинеллиса (Diomidis Spinellis). Ответы Джона Хьюза на мои вопросы по поводу его классической статьи были подробными и понятными. Библиотека реализации паттерна «Посетитель», описанная в завершающей части главы, создана Карин Арну (Karine Arnout) (ныне Карин Безо, Karine Bezault). Я благодарен Глории Мюллер (Gloria Müller) за любопытные наблюдения, которые я почерпнул из ее магистерской диссертации ETH по реализации библиотеки Haskell-подобной функциональности в Eiffel. Я особенно благодарен редакторам этой книги, Диомидису Спинеллису и Георгиосу Гусиосу, за возможность опубликовать эту статью, а также за их исключительное терпение, с которым они относились к задержкам со сдачей материала.

Библиография

Arnout, Karine. 2004. «From patterns to components». Ph.D. thesis, ETH Zurich. <http://se.inf.ethz.ch/people/arnout/patterns/>.

Arnout, Karine, and Bertrand Meyer. 2006. «Pattern componentization: the Factory example». *Innovations in Systems and Software Technology*

(a NASA Journal). New York, NY: Springer-Verlag. <http://www.springerlink.com/content/am08351v30460827/>.

Eber, Jean-Marc, на основании совместной теоретической работы с Саймоном Пейтоном Джонсом (Simon Peyton Jones) и Пьером Вайсом (Pierre Weis). 2001. «Compositional description, baluation, and management of financial contracts: the MLFi language». <http://www.lexifi.com/Downloads/MLFiPresentation.ppt>.

Ecma International. 2006 «*Eiffel: Analysis, Design and Programming Language*». ECMA-367. <http://www.ecma-international.org/publications/standards/Ecma-367.htm>.

Frankau, Simon, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. 2008. «Commercial uses: Going functional on exotic trades». *Journal of Functional Programming*, 19(1):2745, October.

Gamma, Erich, et al. 1994. «*Design Patterns: Elements of Reusable Object-Oriented Software*». Boston, MA: Addison-Wesley.¹

Hughes, John. 1989. «Why functional programming matters.» *Computer Journal*, vol. 32, no. 2: 98–107 (revision of a 1984 paper). <http://www.cs.chalmers.se/~rjmh/Papers/whyfp.pdf>.

La Rochefoucauld, Francois de. 1665. *Réflexions ou sentences et maximes morales*.

Meyer, Bertrand, and Karine Arnout. 2006. «Componentization: the Visitor example». *Computer (IEEE)*, vol. 39, no. 7: 23–30. <http://se.ethz.ch/~meyer/publications/computer/visitor.pdf>.

Meyer, Bertrand. 1992. «*Eiffel: The Language*» (Second Printing). Upper Saddle River, NJ: Prentice Hall.

Meyer, Bertrand. 1997. «*Object-Oriented Software Construction*», Second Edition. Upper Saddle River, NJ: Prentice Hall. <http://archive.eiffel.com/doc/oosc/>.

Meyer, Bertrand. 2004. «The power of abstraction, reuse and simplicity: An object-oriented library for event-driven design». *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, *Lecture Notes in Computer Science* 2635, pp. 236–271. New York, NY: Springer-Verlag. <http://se.ethz.ch/~meyer/publications/lncs/events.pdf>.

Meyer, Bertrand. 2008. «*Touch of Class: An Introduction to Programming Well*». New York, NY: Springer-Verlag. <http://touch.ethz.ch>.

¹ Э. Гамма и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2001–2007.

Peyton Jones, Simon, Jean-Marc Eber, and Julian Seward. 2000. «Composing contracts: An adventure in financial engineering». Functional pearl, in *ACM SIGPLAN International Conference on Functional Programming* (ICFP '00), Montreal, Canada, September '00. ACM Press, pp. 280–292. <http://citeseer.ist.psu.edu/jones00composing.html>.

Peyton Jones, Simon, and Philip Wadler. 1993. «Imperative functional programming». *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Charleston, South Carolina, pp. 71–84. <http://citeseer.ist.psu.edu/peytonjones93imperative.html>.

Steingold, Sam. Online at <http://www.podval.org/?sds/ocaml-sucks.html>.

Sun Microsystems. 1997. «About Microsoft's 'Delegates'». White paper by the Java Language Team at JavaSoft. <http://java.sun.com/docs/white/delegates.html>.

Wadler, Philip. 1995. «Monads for functional programming». *Advanced Functional Programming*, Lecture Notes in Computer Science 925. Eds. J. Jeuring and E. Meijer. New York, NY: Springer-Verlag. <http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf>.

Walden, Kim, and Jean-Marc Nerson. 1994. «Seamless Object-Oriented Software Architecture». Upper Saddle River, NJ: Prentice Hall. http://www.bon-method.com/index_normal.htm.