Компьютер подобен скрипке. Представьте себе новичка, который сначала испытывает проигрыватель, затем скрипку. Скрипка, говорит он, звучит ужасно. Именно этот аргумент мы слышали от наших гуманитариев и специалистов по информатике. Компьютеры, говорят они, хороши для определенных целей, но они недостаточно гибки. Так же и со скрипкой, и с пишущей машинкой, пока Вы не научились их использовать.

Марвин Минский. «Почему программирование — хороший способ выражения малопонятных и туманно сформулированных идей», *Проектирование и планирование*, 1967

Оглавление

Предисловие

Бывает так, что при написании программы вы попадаете в тупик. Возможно, это потому, что вы, как оказалось, не учли некоторые особенности исходной задачи. Однако до обидного часто дело в том, что на начальной стадии проектирования вы приняли какое-то решение, выбрали какую-то структуру данных или способ организации кода, который затем оказался слишком ограниченным, а теперь его трудно изменить.

Эта книга служит мастер-классом по стратегиям организации программ, которые позволяют сохранять гибкость. Все мы уже давно знаем, что, хотя для хранения входных данных легко объявить массив фиксированного размера, такое программное решение может оказаться неприятно ограниченным и привести к тому, что нельзя станет обрабатывать строки больше какой-то длины или наборы записей свыше некоторого фиксированного количества. Многие дыры в безопасности, особенно в сетевом коде, случились потому, что программист выделял буфер фиксированного размера и забывал проверить, что обрабатываемые данные помещаются в этот буфер. Динамически выделяемая память (получаемая от библиотеки в стиле Сишного `malloc` или от автоматического сборщика мусора), будучи несколько сложнее устроена, тем не менее дает намного больше гибкости и, в качестве дополнительного преимущества, позволяет легче избегать ошибок (особенно если язык программирования всегда проверяет обращения к массивам и убеждается, что индекс не выходит за границы). Это всего лишь простейший пример.

Некоторые ранние языки программирования, в сущности, закладывались на способ проектирования, отражающий стиль аппаратной организации, называемый *Гарвардская архитектура*: код программы находится *здесь*, данные — *там*, и задача кода состоит в том, чтобы манипулировать данными. Однако такое жесткое разделение между несмешиваемыми кодом и данными, как оказалось, слишком сильно ограничивает организацию программ. Уже задолго до конца XX в. из опыта функциональных языков программирования (например, ML, Scheme и Haskell) и объектно ориентированных языков (например, Simula, Smalltalk и C++) мы поняли, что у стиля, позволяющего рассматривать данные как код, код как данные и связывать небольшие блоки кода и относящихся к нему данных в единое целое, есть преимущества перед стилем, разделяющим код и данные как отдельные монолитные области. Самый гибкий вид данных — структура-запись, которая может содержать не только «элементарные ячейки» вроде чисел и символов, но и ссылки на выполняемый код, скажем, на функции. Самая мощная разновидность кода — такой, который порождает другой код, объединенный с точно определенным количеством специально отобранных данных; такая связка уже не просто «указатель на функцию», но *замыкание* (в функциональном языке) или *объект* (если язык объектно ориентиро-

ванный).

Джерри Сассман и Крис Хансон используют свой более чем вековой совместный опыт программирования и представляют набор методов, разработанных и проверенных за десятилетия преподавания в Массачусетском технологическом институте, которые позволяют еще более расширить эту базовую стратегию достижения гибкости. Используйте не просто функции, а *полиморфные функции*, открытые для расширения так, как обычным функциям недоступно. Функции должны оставаться маленькими. Часто лучший вариант возвращаемого значения для функции — другая функция (уточненная соответствующими данными). Будьте готовы относиться к данным как к разновидности кода; иногда, если необходимо, это приводит к созданию специализированного встроенного языка внутри вашей программы. (Это один из способов рассказать, откуда взялся язык Scheme: лисповский диалект MacLisp не поддерживал достаточно общий вид замыкания функций, так что мы с Джерри просто написали на нем встроенный вариант Lisp, где такие замыкания были.) Будьте готовы заменить структуру данных структурой более общего вида, которая включает исходную как частный случай и расширяет ее возможности. С помощью автоматического распространения ограничений можно избежать преждевременного решения, какие данные подаются на вход, а какие получаются на выходе.

Эта книга не является обзорной монографией или учебником — скажу повторно, это мастер-класс. В каждой главе можно видеть, как два эксперта демонстрируют тот или иной передовой метод, шаг за шагом разрабатывая работающую подсистему, объясняют на ходу стратегию своей работы и время от времени указывают на подводный камень или способ обойти то или иное ограничение. Будьте готовы, когда вас попросят показать свою способность работать самостоятельно, расширить структуру данных или дописать дополнительный код — а затем подключите свое воображение и идите дальше, чем показали вам авторы. Идеи этой книги богаты и глубоки; внимание как к словам, так и к программам будет вознаграждено.

Гай Стил
Лексингтон, Массачусетс.
Август 2020

Предисловие авторов

Всем нам приходилось тратить уйму лишнего времени за попытками уломать кусок старого кода на выполнение задач, которые мы не представляли себе, когда его исходно писали. Это ужасная трата времени и сил. К сожалению, на нас действует множество сил, побуждающих нас писать программы, хорошо приспособленные к конкретной узкой задаче, где почти нет многократно используемых частей. Однако мы считаем, что так быть не должно.

Строить системы, приемлемо работающие в более широком классе ситуаций, чем их создатели исходно имели в виду, трудно. Лучшие из таких систем способны к развитию: их можно приспособить к новым задачам ценой лишь небольших изменений. Как мы можем проектировать системы, гибкие в этом смысле?

Было бы замечательно, если бы для добавления новой возможности к программе нам нужно было бы только добавить немного кода, без изменения существующего массива программы. Часто этого можно добиться, если использовать определенные организационные принципы при построении этого массива и включать в него соответствующие зацепки.

Многое о построении гибких, способных к развитию систем можно узнать из наблюдения за биологическими системами. Приемы, исходно разработанные для поддержки символического искусственного интеллекта, можно рассматривать как способы увеличить гибкость и приспособляемость программ и других инженерных систем. Напротив, многие практики, принятые в информатике, активно препятствуют построению систем, которые легко перестроить для использования в новых условиях.

Нам часто случалось при написании программ загонять себя в угол и тратить огромные усилия на рефакторинг кода, позволяющий выбраться из тупика. Поэтому мы считаем, что набрали достаточный опыт и способны выявить, определить и продемонстрировать найденные нами стратегии и методы, эффективные при построении больших систем, способных адаптироваться к задачам, которые создатели исходно не имели в виду. В этой книге мы делимся некоторыми плодами своего более чем векового опыта программирования.

Эта книга

Эта книга появилась в процессе преподавания компьютерного программирования в Массачусетском технологическом институте (MIT). Мы начали вести этот курс много лет назад, желая показать старшекурсникам и аспирантам методы и технологии, полезные при решении основных задач в области искусственного интеллекта, таких как символическая математика и системы, основанные на правилах. Мы хотели,

чтобы студенты были способны строить эти системы гибко, чтобы их можно было легко сочетать при построении еще более мощных программ. Кроме того, мы хотели обучить студентов рассуждать о зависимостях — как их отслеживать, использовать для объяснения поведения и как с их помощью управлять перебором.

Несмотря на то что наш курс был и остается успешным, оказалось, что в начале мы понимали материал не настолько хорошо, как сами об этом думали. Поэтому пришлось потратить немало сил на полировку своих инструментов и уточнение идей. Теперь мы понимаем, что эти методы работают далеко не только в области искусственного интеллекта. Мы считаем, что нашим опытом может воспользоваться любой, кто строит сложные системы, например компиляторы и интегрированные среды разработки. Книга построена на основе лекций и упражнений, которые мы сейчас используем в своем курсе.

Содержание

В этой книге намного больше материала, чем можно пройти за односеместровый курс. Поэтому каждый год мы отбираем некоторую часть для использования в классе. Глава 1 представляет собой введение в нашу философию программирования. Здесь мы демонстрируем понятие *гибкости* в общем контексте природы и техники. Мы стараемся показать, что гибкость столь же важна, как эффективность и правильность поведения. В каждой последующей главе мы вводим некоторый общий метод и иллюстрируем его набором упражнений. Это важный организующий принцип всей книги.

В главе 2 исследуем некоторые универсально применимые способы построения систем, способных к росту. Мощный метод организации гибкой системы — построить ее в виде набора языков специального назначения, каждый из которых приспособлен для описания некоторой подсистемы. Здесь мы разрабатываем основной инструментарий для построения специализированных языков: показываем, как подсистемы можно организовать из взаимозаменяемых частей, как их можно гибко соединять с помощью *комбинаторов*, как части системы обобщаются через *обертки* и как программу часто можно упростить, абстрагируя модель предметной области.

В главе 3 мы вводим чрезвычайно мощный, хотя потенциально опасный метод использования *полиморфных процедур*, управляемых предикатами. В начале мы строим обобщение арифметики для работы с символьными алгебраическими выражениями. Затем показываем, как такое обобщение можно сделать эффективным с помощью меток типа на данных, и демонстрируем силу нашего метода на примере простой, но легко расширяемой игры-бродилки.

В главе 4 мы вводим понятие *сопоставления с образцом*, сначала для использования в системах переписывания термов, затем через *унификацию* показываем, как легко можно построить вывод типов. Сегментные переменные заставляют нас использовать *перебор с возвратами*. Унификация — первый случай, где мы видим силу представления и комбинирования структур с *частичной информацией*. В конце главы мы расширяем эту идею на задачу сопоставления в произвольных графах.

В главе 5 мы исследуем возможности *интерпретации* и *компиляции*. Мы считаем, что программисты должны быть способны выйти за рамки своего языка программирования, каков бы он ни был, и использовать язык, лучше приспособленный для решения конкретной задачи. Мы также показываем, как естественным образом встроить в язык перебор с возвратами путем реализации в составе интерпре-

татора/компилятора оператора недетерминистского выбора `amb` и как использовать *продолжения*.

В главе 6 мы показываем построение систем *многослойных данных* и *многослойных процедур*, где к каждому элементу данных могут в качестве аннотации быть прикреплены метаданные различного рода. Метаданные никак не влияют на обработку нижележащих данных, и код для обработки нижележащих данных не обязан даже знать о них и на них ссылаться. Однако метаданные обрабатываются своими собственными процедурами, в сущности, параллельно основным данным. В качестве иллюстрации мы присоединяем к числовым данным информацию о единицах измерения, а также демонстрируем отслеживание зависимостей, так что каждый элемент данных оказывается снабжен родословной, показывающей, как он получен из элементарных источников.

Все это собирается вместе в главе 7, где мы вводим *распространение*, чтобы освободиться от парадигмы компьютерных языков, ориентированной на выражения. Здесь мы смотрим на соединение модулей как на монтажную схему. Это позволяет нам гибко подключать различные источники частичной информации. Поскольку мы отслеживаем зависимости с помощью многослойных данных, то способны организовать *перебор, управляемый зависимостями*, который значительно уменьшает пространство поиска в больших и сложных системах.

С помощью этой книги можно построить несколько вариантов университетского курса продвинутого уровня. Идея комбинаторов из главы 2 и полиморфные процедуры из главы 3 используются во всех последующих главах. Однако образцы и сопоставление из главы 4, а также вычислители из главы 5 в дальнейших главах не встречаются. Единственный материал из главы 5, необходимый в дальнейшем, — оператор `amb` из разделов 5.4 и 5.4.1. Идея многослойности из главы 6 близко связана с идеей полиморфных процедур, но с некоторым новым поворотом. Использование многослойных данных для отслеживания зависимостей, которое в главе 6 служит примером, становится важным компонентом системы в работе над распространением в главе 7, где зависимости служат для оптимизации поиска с возвратами.

Scheme

Программы в этой книге написаны на Scheme, функциональном по преимуществу языке, который является вариантом Lisp. Несмотря на то что Scheme не популярный язык и не используется широко в промышленном программировании, для этой книги он — правильный выбор[1]. Цель этой книги — демонстрация и объяснение идей. По многим причинам представление кода, воплощающего и иллюстрирующего эти идеи, на Scheme оказывается короче и проще, чем на более популярных языках. Некоторые из наших идей на других языках было бы почти невозможно выразить.

Языки из других семейств, кроме Lisp, требуют множества церемоний, чтобы сказать простые вещи. В нашем же случае единственное, что удлинняет наш код, — описательные имена для вычислительных объектов.

Благодаря тому, что синтаксис Scheme чрезвычайно прост — это всего лишь представление естественного дерева разбора, и требуется лишь минимальный синтаксический анализ, — легко оказывается писать программы, работающие с текстами других программ, такие как интерпретаторы, компиляторы и обработчики алгебраических выражений.

[1] Мы даем краткое введение в Scheme в приложении B.

Важно и то, что Scheme — язык снисходительный, а не нормативный. Он не пытается запретить программисту делать что-то «глупое». Это позволяет нам проделывать некоторые мощные трюки, например динамически перестраивать значение арифметических выражений. В языке, накладывающем на программиста более строгие правила, нам бы это не удалось.

Scheme разрешает использовать присваивание, но поощряет функциональное программирование. Язык Scheme не содержит статических типов, но в нем очень строгая динамическая типизация, что позволяет иметь безопасное выделение памяти и сборку мусора: пользовательская программа не может создать указатель из ничего или получить доступ к произвольной ячейке памяти. Тут дело не в том, что мы считаем статическую типизацию плохой идеей. Она, безусловно, полезна для изгнания многих видов ошибок из программы на раннем этапе. Более того, система типов вроде той, что в языке Haskell, может быть полезна при продумывании стратегий программирования. Однако в этой книге интеллектуальный вес статических типов помешал бы рассматривать потенциально опасные стратегии повышения гибкости.

Кроме того, Scheme имеет некоторые особые свойства, например явную поддержку продолжений и динамического связывания, которых нет в большинстве языков. Эти свойства позволяют нам реализовать такие мощные механизмы, как недетерминистский оператор `amb`, в самом языке (не используя отдельный уровень интерпретации).

Глава 1

Гибкость в природе и в проектировании

Спроектировать общеполезный механизм, который будет хорошо справляться с любой работой, очень трудно. Поэтому большинство инженерных систем предназначено для выполнения какой-то конкретной задачи. Изобретения общего назначения, такие как винт с резьбой, случаются редко и оказывают большое влияние. В этом отношении цифровой компьютер является прорывом, поскольку это универсальная машина, способная сымитировать любое другое устройство для обработки информации[1]. Мы пишем программы, которые конфигурируют наш компьютер, чтобы он выполнял такую имитацию с прицелом на конкретные нужные нам задачи.

Мы научились разрабатывать программные системы, отлично решающие конкретные задачи, в качестве расширения обычной инженерной практики прошлых лет. Каждый фрагмент программы предназначен для выполнения некоторой относительно узкой задачи. Если решаемая задача изменяется, требуется модифицировать и программу. Однако не всегда небольшие изменения в задаче приводят лишь к небольшим изменениям в программе. Программа слишком плотно подогнана под конкретный вид работы, чтобы оставалось место для гибкости. Вследствие этого программные системы оказываются неспособны развиваться, сохраняя изящество. Они хрупки, и, если область приложения изменяется, такой проект требуется заменить на совершенно новый[2]. Выходит медленно и дорого.

Инженерные системы не обязательно должны быть хрупкими. Например, интернет выдержал рост от небольшой системы до глобальной. Города способны органично развиваться, воспринимая новые модели бизнеса, стили жизни, виды транспорта и связи. Глядя на биологические системы, мы видим, что возможно построить структуры, приспосабливающиеся к изменениям в окружении, как индивидуально, так и

[1] Открытие универсальных машин Тьюрингом [124], а также то,что множество функций, вычислимых машинами Тьюринга, эквивалентно как множеству функций, представимых в λ-исчислении Алонсо Чёрча [17, 18, 16], так и общерекурсивным функциям Курта Гёделя [45] и Жака Эрбрана [55], — одно из величайших интеллектуальных достижений XX в.

[2] Разумеется, существуют замечательные исключения. Например, *Emacs* — расширяемый редактор кода, изящно приспособившийся как к изменениям в вычислительном окружении, так и к изменениям в ожиданиях пользователя. В мире информатики только начинают исследовать «инженерные фреймворки», например `.net` от Микрософт или Java компании Sun. Такие конструкции предназначены быть инфраструктурой, поддерживающей развивающиеся системы.

в качестве эволюционного ансамбля. Почему же большинство программ не проектируется и не пишется таким образом? На то есть исторические причины, однако главная причина состоит в том, что мы не знаем, как этого добиться в общем случае. До сих пор если система оказывается способной выдержать изменение в требованиях к ней, то это лишь счастливое совпадение.

Аддитивное программирование

Нашей целью в этой книге будет исследовать, как можно строить вычислительные системы, которые можно легко приспособить к новым задачам. Работающую программу не следует модифицировать. Должно быть возможным добавить к ней реализацию новой функциональности либо подстроить старые функции под новые требования. Мы называем такую цель *аддитивным программированием*. Мы будем исследовать методы, позволяющие добавить функциональность к существующей программе, не ломая ее. Это не гарантирует, что добавления будут правильно работать: их тоже надо отлаживать; однако они не должны нечаянно разрушать существующую функциональность.

Многие из методов, исследуемых нами в этой книге, не новы: некоторые существуют с самых ранних лет работы с компьютерами! Они также не составляют единого согласованного набора, это просто коллекция приемов, которые кажутся нам полезными. Мы не столько хотим рекомендовать использование этих конкретных методов, сколько стимулировать стиль мышления, направленный на гибкость.

Для того чтобы сделать аддитивное программирование возможным, необходимо минимизировать предположения о том, как программа работает и как она будет использоваться. Решения, принятые на этапе проектирования и написания программы, могут сузить ее будущие возможные расширения. Вместо того чтобы делать такие предположения, мы строим программы так, чтобы решения принимались точно вовремя на основе конкретного окружения, где им предстоит работать. Мы рассмотрим несколько приемов, поддерживающих такой стиль проектирования.

Программы всегда можно сочетать так, чтобы получилось объединение режимов поведения, обеспечиваемых каждой из них. Однако нам хочется, чтобы целое было больше, чем сумма его частей; нужно, чтобы части общей системы сотрудничали и чтобы в результате появлялись возможности, которые ни одна из компонент сама по себе предоставить не может. Однако здесь необходимы компромиссы: части, из которых мы составляем систему, должны четко разделять зоны ответственности. Если модуль программы хорошо решает одну задачу, его проще заново использовать и проще отлаживать, чем такой, который объединяет несколько различных задач. Если нам требуется аддитивное построение, важно, чтобы отдельные части сочетались с минимумом нежелательных взаимодействий.

Для достижения аддитивного программирования необходимо, чтобы компоненты нашей системы были как можно проще и обладали максимальной общностью. К примеру, компонента, принимающая более широкое множество возможных входных данных, чем строго необходимо для исходно поставленной задачи, будет применима в большем числе случаев, чем та, где этим не озаботились. Семейства модулей, построенных на основе стандартизированного интерфейса, можно сочетать и смешивать, получая великое множество разных систем. Важно также правильно выбрать уровень абстракции наших компонент, выделив для них общую предметную область, и затем построить семейство, поддерживающее эту предметную область. Мы начнем

рассматривать эти требования в главе 2.

Чтобы добиться максимальной гибкости, множество порождаемых значений компоненты должно быть как можно уже и как можно лучше определено — намного меньше, чем множество приемлемых входов для любого модуля, который будет считывать эти данные. Это аналогично статической дисциплине в цифровой абстракции, которой мы обучаем студентов на вводных курсах по вычислительным системам [126]. Сущность цифровой абстракции состоит в том, что выход предыдущего модуля всегда должен быть лучше, чем приемлемое качество входа следующего, так что шум подавляется.

В программной инженерии этот принцип известен как «закон Постела», названный в честь одного из основателей интернета Джона Постела. Он писал в RFC760 [97], описывающем протокол IP: «Реализация протокола должна быть гибкой и разумной. Каждая реализация должна предполагать интероперабельность с продукцией других разработчиков. Хотя целью данной спецификации является четкое и строгое описание протокола, существует вероятность различных интерпретаций стандарта. При передаче дейтаграмм следует строго следовать спецификации, сохраняя в то же время готовность к восприятию любых дейтаграмм, которые можно интерпретировать»*. Обычно это формулируется так: «Будь консервативным в собственном поведении, но либеральным при восприятии чужого».

При использовании модулей, приспособленных к более общим задачам, чем кажется строго необходимым, вся структура нашей системы приобретает дополнительную гибкость. Она выдерживает небольшие нарушения исходных требований, поскольку каждая компонента готова воспринять нарушенные (шумные) входные данные.

Семейство компонент, способных сочетаться друг с другом, служит основой *специализированного языка* для некоторой предметной области. Часто наилучшим способом решить сразу несколько сложных задач является создание нового языка — набора элементарных конструкций, средств их сочетания и абстракции, — который облегчает формулировку решений для этих задач. Поэтому мы хотим научиться производить специализированные языки по необходимости, а также гибко сочетать их. Мы начинаем рассматривать специализированные языки в главе 2. Еще более мощным средством является реализация таких языков методом прямого вычисления. Эта идея рассматривается в главе 5.

Одна из стратегий увеличения гибкости, знакомая многим программистам, называется *полиморфные вызовы*. Мы тщательно исследуем ее в главе 3. Полиморфный вызов часто позволяет расширить область применимости процедуры путем добавления дополнительных обработчиков (методов) в зависимости от того, какие аргументы передаются процедуре. Если мы потребуем, чтобы множества аргументов, на которые реагируют обработчики, не пересекались, мы избежим поломки программы при добавлении каждого нового обработчика. Однако, в отличие от полиморфного вызова в типичном объектно ориентированном контексте, наша концепция полиморфизма не предполагает таких понятий, как классы, экземпляры или наследование. Эти понятия ослабляют разграничение задач, поскольку вносят излишние онтологические обязательства.

Совершенно другая стратегия, рассматриваемая в главе 6, — разделение *слоев* как данных, так и процедур. Здесь используется идея, что данные обычно снабжены метаданными, которые можно обрабатывать отдельно от них. К примеру, числовые

*Перевод Н. Малых. — *Прим. перев.*

данные обычно ассоциируются с единицами измерения. Мы увидим, как гибкое добавление слоев к готовой программе может расширить ее функциональность без всякого изменения исходного кода.

Кроме того, возможно построение систем, сочетающих различные источники *частичной информации* и получающих более полные результаты. Это работает особенно хорошо, если вклады разных источников независимы друг от друга. В главе 4 мы увидим, что вывод типов сводится к сочетанию различных источников частичной информации. Локально выводимые свидетельства о типе значений, например что численное сравнение требует числа на входе и получает на выходе булево значение, можно скомбинировать с другими локальными ограничениями и получить нелокальные ограничения.

В главе 7 мы увидим еще один способ сочетания частичной информации. Расстояние до соседней звезды можно оценить геометрическим методом параллакса: измерить угол, на который образ звезды сдвигается относительно более далеких объектов, когда Земля вращается вокруг Солнца. Расстояние до звезды также можно оценить, рассматривая ее яркость и спектр, используя наши знания о структуре и эволюции звезд. Эти оценки можно сравнить друг с другом и получить новую оценку, которая будет точнее, чем сделанная каждым отдельным методом.

Двойственная идея состоит в *дублировании*, когда есть несколько способов получения некоторых данных, которые мы можем комбинировать и сравнивать. Существует множество способов использовать дублирование, включая поиск ошибок, управление производительностью и распознавание несанкционированного доступа. Важно также то, что система с дублированием аддитивна: каждая ее компонента самодостаточна и может получить результат сама по себе. Один из интересных способов использовать дублирование состоит в том, чтобы динамически выбирать среди нескольких реализаций алгоритма в зависимости от контекста. При этом можно избежать предположений о том, как именно будет использоваться каждая отдельная реализация.

Проектирование и встраивание гибкости в систему влечет определенные затраты. Процедура, способная воспринять больше видов данных, чем необходимо для конкретной задачи, будет содержать больше кода, чем нужно для решения прямо сейчас, и потребует от программиста больше размышлений, чем абсолютно необходимо прямо сейчас. То же самое относится к полиморфным вызовам, многослойной организации и дублированию; каждое из них требует постоянных затрат памяти, времени вычислений и/или времени работы программиста.

Однако основные затраты на программное обеспечение — усилия, затраченные программистами за все время жизни продукта, включая поддержку и адаптации, необходимые при изменяющихся требованиях. Методы проектирования, которые минимизируют переписывание и рефакторинг, снижают общую стоимость, сводя ее к последовательным добавкам функциональности вместо полной переработки. Другими словами, долговременные затраты становятся аддитивными вместо мультипликативных.

1.1 Архитектура вычислений

Для систем того рода, что мы имеем в виду, может быть полезна архитектурная метафора. После того как исследованы природа участка, на котором предполагается строить, и требования к возводимой постройке, процесс проектирования начинается

с *эскизного проекта*: принципов организации работы[3]. Как правило, эскизный проект представляет собой набросок геометрического расположения частей здания. Он может также выражать некоторые абстрактные идеи, например разделение между «рабочими пространствами» и «обслуживающими пространствами», как в работах Луиса Исидора Кана [130]. Такая декомпозиция служит для того, чтобы разделить архитектурную задачу на части, отделив инфраструктурную поддержку вроде коридоров, туалетов, серверных или лифтов, от пространств, которые нужно поддерживать, таких как лаборатории, учебные классы и офисы учебного здания.

Эскизный проект является моделью, однако обычно это еще не полностью работоспособная структура. Требуется развить его, дополнив функциональными элементами. Где проходят воздуховоды вентиляции, водопроводные трубы, системы распределения электричества и связи? Где проложить дорогу, чтобы обеспечить удобство доставки грузов и обслуживание постройки? Эти уточнения могут потребовать некоторых изменений изначального эскиза, однако он продолжает служить как каркас, на который опираются конкретные решения.

В программировании эскизный проект — это абстрактный план вычисления, которое надо произвести. В небольшом масштабе это может быть абстрактный алгоритм или описание структуры данных. В более крупных системах эскиз может содержать абстрактную композицию фаз и параллельных ветвей вычисления. В еще более крупных проектах в него может входить распределение задач по логическим (или даже физическим) локациям.

Исторически сложилось так, что у программистов обычно нет возможности строить эскизные проекты подобно архитекторам. В подробных языках вроде Java эскиз тесно перемешан с проработкой деталей. «Рабочие пространства», выражения, напрямую описывающие требуемое поведение, никак не отделены от «обслуживающих пространств» вроде объявлений типов и классов, а также импорта и экспорта библиотек[4]. Более свободные языки, такие как Lisp и Python, почти не оставляют места для обслуживающих пространств, и попытки добавить к ним объявления типов, даже в виде аннотаций, встречают сопротивление, поскольку они затемняют красоту и ясность эскиза.

В архитектуре эскизный проект должен быть достаточно полон, чтобы на его основе строить модели, доступные для анализа и критики. Скелетная часть программы также должна быть достаточно адекватна для анализа и критики, но она также должна быть исполнима, чтобы производить с ней эксперименты и отлаживать. Так же, как архитектор должен дополнить эскизный проект, чтобы полностью реализовать проектируемую структуру, программист должен дополнить базовый план, чтобы воплотить требуемую вычислительную систему. Многослойная разработка (описываемая нами в главе 6) — один из способов построения систем, позволяющих такую постепенную детализацию.

[3]Эскизный проект представляет собой основную идею архитектурного произведения: это «представленная в целом композиция, детали которой должны быть проработаны позже».[62]

[4]В Java есть интерфейсы, которые можно рассматривать как разновидность эскизного проекта, поскольку они являются абстрактным представлением программы. Однако в архитектуре эскизный проект сочетает абстрактные и конкретные компоненты, а интерфейсы языка Java целиком абстрактны. К тому же многие программисты полагают, что чрезмерное использование интерфейсов «плохо пахнет».

1.2 Гибкость через умные компоненты

Большие системы строятся из множества компонент меньшего размера, и каждая из них участвует в выполнении общей задачи, либо напрямую решая ее часть, либо через взаимодействие с другими компонентами по схеме, определенной архитектором системы. В проектировании систем центральной проблемой является разработка интерфейсов, позволяющих связывать компоненты, чтобы функции этих компонент сочетались в построении составной функции.

В относительно простых системах системный архитектор может указать формальные спецификации различных интерфейсов, которые производители соединяемых компонент должны выполнить. Поразительный успех электроники основывается на том, что в этой области возможно составить такие спецификации и выполнить их. Высокочастотные аналоговые устройства связываются между собой с помощью коаксиальных кабелей со стандартизованным сопротивлением и стандартизованными семействами разъемов [4]. Как назначение устройства, так и поведение его интерфейса обычно можно описать с помощью лишь нескольких параметров [60]. В цифровых системах все еще проще: имеются статические описания значений сигналов (цифровая абстракция), динамическое описание временных характеристик сигналов [126], и, наконец, есть механические описания форм-факторов компонент[5].

К сожалению, подобная априорная спецификация становится все труднее по мере возрастания сложности систем. Можно указать, что программа для игры в шахматы должна играть *по правилам* — не должна их нарушать, — но как нам удастся задать требование, чтобы она играла в шахматы *хорошо*? Программные системы строятся из большого количества уникальных, сильно специализированных частей. Сложность спецификации программных компонент усиливается от того, что многие из них обладают индивидуальными характеристиками.

В противоположность этому биология способна создавать системы умопомрачительной сложности без особенно больших описаний (если даже считать проблему описаний решенной!). Каждая клетка нашего тела происходит от единственной зиготы. Все клетки содержат идентичную наследственность (около 1 Гб постоянной памяти!). Однако существуют клетки кожи, нейроны, мускульные клетки и т. д. Клетки самоорганизуются в ткани, органы и системы органов. В итоге 1 Гб ПЗУ описывает, как построить чрезвычайно сложную машину (человека) из большого числа ненадежных компонент. В этой информации задается, как работают эти компоненты и как их конфигурировать. Кроме того, описывается, как надежно управлять этой машиной в разнообразных неблагоприятных обстоятельствах в течение долгого времени жизни и как эту машину защищать от других, которые бы хотели ее съесть!

Если бы наши программные компоненты были проще или более широко применимы, их спецификации тоже были бы проще. Если бы компоненты умели при-

[5] *Книга данных TTL для инженеров-проектировщиков*[123] может служить классическим примером набора спецификаций компонент для цифровых систем. В TTL описываются несколько внутренне согласованных «семейств» интегральных схем малого и среднего масштаба. Эти семейства различаются своими скоростными и температурными характеристиками, но не функциями. Спецификация указывает статические и динамические характеристики каждого семейства, функции, доступные в каждом семействе, и физические параметры упаковки компонент. Компоненты согласованы не только внутренне, но и между собой, т. е. каждая функция доступна в каждом семействе, в одинаковой упаковке и с единой номенклатурой описания. Таким образом, инженер может спроектировать некоторую составную функцию и только затем выбрать семейство для ее реализации. Всякий хороший инженер (и даже биолог!) должен усвоить уроки TTL.

способливаться к окружению, точность спецификаций была бы не так важна. При помощи обеих этих стратегий биологическим системам удается построить надежные сложные организмы. Разница состоит в том, что биологические клетки конфигурируются динамически и могут приспосабливаться к контексту. Это оказывается возможным, поскольку то, как клетка дифференцируется и специализируется, зависит от ее окружения. Обычно наши программы не обладают этим свойством, и поэтому нам приходится настраивать каждый их модуль вручную. Как же в принципе способна работать биология?

Рассмотрим еще один пример. Известно, что различные части мозга соединяются огромными пучками нервов, и в геноме даже близко не достаточно информации, чтобы во всех деталях описать эти связи. Вероятно, различные части мозга учатся взаимодействовать друг с другом, основываясь на сходстве своего опыта[6]. Таким образом, интерфейсы должны самоконфигурироваться на основании некоторых правил непротиворечивости, информации о среде и активного исследовательского поведения. Это имеет высокую цену во время начальной загрузки (конфигурация полноценного человека занимает несколько лет), но в результате система обладает устойчивостью, недостижимой для наших современных инженерных систем.

Одна из идей состоит в том, что биологические системы используют не императивные, а информативные контекстные сигналы[7]. Не существует управляющего блока, который говорит, что делать каждой из частей; вместо этого части самостоятельно выбирают роли на основе своего окружения. Поведение клеток не кодируется в сигналах; оно отдельно выражено в геноме. Комбинации сигналов только включают одни модели поведения и отключают другие. Такое слабое связывание обеспечивает вариативность способов поведения, срабатывающих в различных точках организма, без того, чтобы изменять механизм, определяющий сами эти точки. Организованные таким образом системы способны к развитию, поскольку они могут запускать адаптивное варьирование в некоторых точках, не меняя поведение подсистем в других точках.

Традиционные программные системы строятся по императивной модели, где существует иерархия управления, однозначно определяемая структурой. Отдельные части системы играют роль безвольных подчиненных, которые делают то, что им сказано. Адаптация в таких системах становится сложной, потому что любые изменения должны отражаться во всей структуре управления. В общественных системах нам известно, какие проблемы происходят из-за строгих структур власти и централизованного управления. Однако наши программы следуют этой порочной модели. Можно сделать лучше: если наши компоненты будут умнее и при этом будут по отдельности отвечать за свое поведение, им будет проще приспосабливаться, поскольку на каждое изменение придется реагировать только тем частям системы, которых оно непосредственно касается.

Планы строения

Все позвоночные имеют почти одинаковый план строения, однако вариация в его деталях громадна. Действительно у всех животных с двусторонней симметрией имеются одни и те же гомеобоксные гены, такие как комплекс Нох. Эти гены создают

[6] Простейшую версию такого самоконфигурирующегося поведения продемонстрировал Джейкоб Бил в магистерской диссертации [9].

[7] Ее исследуют Киршнер и Герхарт [70].

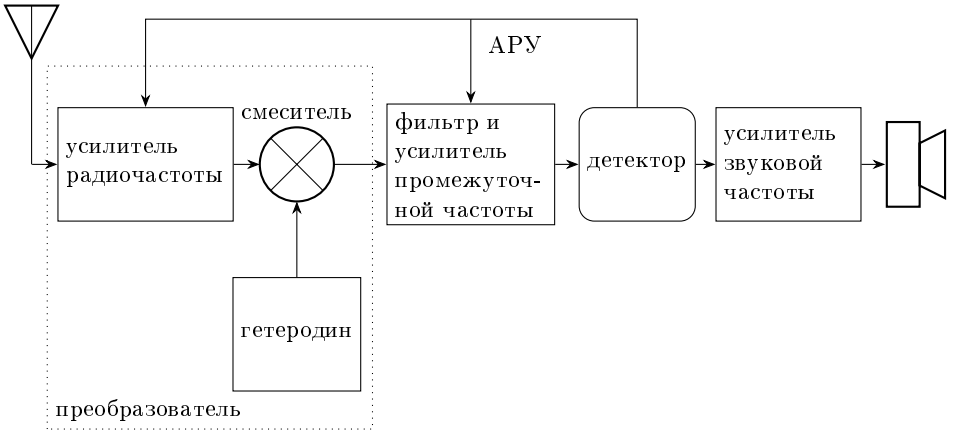

Рис. 1.1. Супергетеродинная схема, которую в 1918 году изобрел майор Эдвин Армстронг, по-прежнему остается преобладающим «планом строения» радиоприемника

приблизительную систему координат при развитии животного, разделяя формирующееся тело на различные области[8]. Каждая область служит контекстом для дифференциации клеток. Информация, выводимая из контакта с соседними клетками, создает дополнительный контекст, на основе которого выбирается конкретное поведение среди моделей возможного поведения, заложенных в генетическую программу клетки[9]. Даже методы построения тканей используются одни и те же — морфогенез проточных желез и органов типа легких и почек основываются на одном эмбриологическом приеме: внесение эпителия в мезенхиму автомагически[10] порождает ветвящийся лабиринт слепых трубочек, окруженных дифференцирующейся мезенхимой[11].

Качественная инженерная работа отличается тем же стилем: хороший проект всегда разбивается на модули. Рассмотрим устройство радиоприемника. Было изобретено несколько общих «планов строения»: приемник прямого преобразования, прямого усиления и супергетеродинный. Каждый из них содержит последовательность локаций, определяемых инженерным эквивалентом комплекса Нох, который накладывает шаблон на систему от антенны до выходного преобразователя. Например, супергетеродинный приемник (рис. 1.1) разбивается на стандартные локации, от носа до хвоста.

Каждый из модулей, указанных на этом плане, сам делится на более мелкие модули — генераторы сигналов, смесители, фильтры и усилители и т. д. вплоть до отдельных электронных деталей. Кроме того, каждый модуль может быть реализован несколькими различными способами: блок радиочастоты может быть просто

[8] Это лишь приблизительно описывает сложный процесс, включающий градиенты морфогенов. Мы здесь не стремимся к большей точности, поскольку речь идет не о биологии, а о том, как знание биологии может помочь инженеру.

[9] Мы исследовали некоторые вопросы программирования при развитии такого рода в проекте Аморфных вычислений [2].

[10] Автомагически: «Автоматически, но таким способом, который автор почему-то (как правило из-за чрезмерной сложности, или чрезмерного уродства, или даже тривиальности) не считает нужным объяснять». Из *Словаря хакера* [117, 101].

[11] Один из хорошо изученных примеров такого механизма — формирование подчелюстной железы у мыши. См., например, исследование [11] или краткое описание в [7], раздел 3.4.3.

фильтром, а может представлять собой сложную комбинацию фильтра и усилителя. В аналоговом телевизионном приемнике часть выхода смесителя обрабатывается видеоцепью как сигнал с амплитудной модуляцией, а часть — аудиоцепью как сигнал с частотной модуляцией. Некоторые блоки, например преобразователь, могут разворачиваться рекурсивно (как если бы часть Нох-комплекса была продублирована!), так что получаются приемники с множественным преобразованием.

В биологических системах структура разделения блоков поддерживается и на более высоких уровнях организации. Имеются ткани, специально предназначенные разделять отделы тела, и трубки, соединяющие их. Органы разграничиваются такими тканями и связываются этими трубками, а вся структура упакована в целомы — в высших организмах это полости, выстеленные специальными тканями.

Подобные же приемы можно применять и в построении программ. План строения — это всего лишь обертка, соединяющая частично специфицированные компоненты. Это разновидность *комбинатора*: нечто, связывающее блоки в более крупный блок. Можно создавать *комбинаторные языки*, где компоненты и составленное из них целое имеют одну и ту же спецификацию интерфейса. В комбинаторном языке можно создать композитные блоки произвольного размера, сочетая между собой небольшое количество видов элементарных компонент. Самоподобные структуры облегчают построение композитов. Мы начнем строить программы на основе комбинаторов в главе 2, а затем эта тема будет сопровождать нас на протяжении всей книги.

Нечто подобное можно сделать с помощью специализированных языков. Абстрагировав предметную область, можно применять один и тот же предметно-независимый код в разных областях. Например, численные интеграторы полезны всюду, где используются числа, независимо от предметной области. Другим примером может служить сопоставление с образцом из главы 4, которое также применимо в большом количестве ситуаций.

Биологические механизмы универсальны в том смысле, что, в принципе, любая компонента может работать как любая другая. Аналоговые электронные компоненты не являются в этом смысле универсальными. Они не могут приспосабливаться к своему окружению на основе локальных сигналов. Однако существуют универсальные электронные строительные блоки (например, программируемый компьютер с аналоговыми интерфейсами!)[12]. Для низкочастотных применений можно строить аналоговые системы из таких блоков. Если бы в каждом блоке содержался весь код, требуемый для построения любого блока системы, но он бы получал специализацию путем взаимодействия с соседями, и если бы в общем наборе были неспециализированные «стволовые клетки», мы могли бы представить самоконфигурирующиеся и саморемонтирующиеся аналоговые системы. Но в наше время мы пока что проектируем и строим части системы по отдельности.

В программировании у нас есть понятие универсального элемента: *вычислитель*. Вычислитель принимает на вход описание некоторого подлежащего исполнению вычисления и входные данные для него. Он выдает выходные значения, которые были бы получены, если бы входные данные были скормлены специализированному компоненту, предназначенному выполнять это вычисление. В области вычислений у нас есть возможность следовать мощной в своей гибкости стратегии эмбрионального развития. Мы подробнее рассмотрим использование стратегии вычислителя в

[12]Петр Митрос разработал новую стратегию проектирования для построения аналоговых цепей из потенциально универсальных строительных блоков. См. [92].

главе 5.

1.3 Избыточность и дублирование

Биологические системы обладают немалой прочностью. Одна из характеристик биологических систем — они всегда *избыточны*. Органы вроде печени или почек обладают этой избыточностью в большой степени: их мощность значительно превосходит необходимый для жизни минимум, так что человек без одной из почек или части печени не теряет в качестве жизни. Кроме того, в биологических системах развито *дублирование*: как правило, любое требование можно выполнить множеством различных способов[13]. Например, если поврежден палец, можно сложить остальные пальцы и схватить ими объект. Необходимую для жизни энергию можно получить из множества различных видов сырья: метаболизму подвергаются углеводы, жиры и белки, хотя механизмы пищеварения и извлечения энергии для каждого из этих источников сильно различаются.

Генетический код сам по себе обладает свойством дублирования, поскольку кодоны (тройки нуклеотидов) отображаются на аминокислоты не один к одному: 64 возможных кодона задают всего лишь около 20 возможных аминокислот [86, 54]. В результате многие точечные мутации (замены одного нуклеотида) не изменяют белок, задаваемый кодирующей областью. Кроме того, замена одной аминокислоты на похожую часто не ухудшает биологическую активность белка. Такое дублирование обеспечивает способы накопления изменчивости без очевидных фенотипических последствий. Более того, если сам ген дублируется (что тоже случается нередко), его копии могут молчаливо разойтись, позволяя таким образом развиться вариантам, которые могут оказаться полезны в будущем, без влияния на текущую жизнеспособность организма. Вдобавок копии могут оказаться под управлением различных механизмов транскрипционного контроля.

Дублирование является результатом эволюции, но оно и помогает эволюции. Скорее всего, дублирование само по себе поддерживается отбором, поскольку только существа со значительным дублированием обладают достаточной приспособляемостью, чтобы выжить при изменениях среды обитания[14]. Допустим, например, что у нас есть некоторое существо (или инженерная система), обладающее дублированием в том смысле, что для некоторой существенной функции у него есть несколько различных независимых механизмов. Если среда обитания (или технические требования) изменится так, что один из способов исполнения этой функции перестанет работать, существо продолжит жить и размножаться (система продолжит работать согласно спецификации). Однако подсистема, которая стала нерабочей, теперь доступна для мутации (или ремонта) без влияния на жизнеспособность (текущую работоспособность) системы в целом.

Теоретическое строение физики глубоко дублировано. Например, задачи классической механики могут решаться несколькими способами. Существует ньютоновская формулировка векторной механики и лагранжевская или гамильтоновская формулировка вариационной механики. Если применимы и ньютоновская, и какая-либо из форм вариационной формулировки, они дают эквивалентные уравнения движения.

[13] Различие, проводимое биологами между избыточностью и дублированием, ясно проявляется в типичных случаях, но в пограничных условиях расплывается. Более подробно см. [32].

[14] Специалисты по информатике исследовали эволюцию приспособляемости методами имитационного моделирования [3].

Для анализа систем с диссипативными силами (такими, как трение) эффективна векторная механика; для таких систем вариационные методы плохо приспособлены. Лагранжева механика работает намного лучше, чем векторная, для систем с жесткими ограничениями, а гамильтонова механика дает канонические преобразования, которые позволяют лучше понять системы с помощью структуры фазового пространства. Как лагранжевская, так и гамильтоновская формулировки отражают глубокую интуицию по поводу роли симметрий и законов сохранения. То, что существует три перекрывающихся способа описания механической системы, согласованных между собой в тех случаях, когда все они применимы, позволяет решать каждую конкретную задачу наиболее удобным образом [121].

Инженерные системы могут обладать некоторой избыточностью в случае критически важных систем, где цена сбоя очень высока. Однако они почти никогда не включают намеренное дублирование такого типа, как в биологических системах, кроме как в виде побочного результата неоптимального проектирования[15].

Дублирование может приносить пользу в наших системах: как и в случае избыточности, можно сравнивать результаты дублированных вычислений, что повышает надежность. Однако дублированные вычисления не просто избыточны, они *отличаются* друг от друга, а значит, программная ошибка в одном из них, скорее всего, не влияет на остальные. Это положительно влияет не только на надежность, но и на безопасность, поскольку успешная атака должна будет скомпрометировать несколько продублированных модулей.

В случае, когда дублированные модули порождают частичную информацию, комбинация результатов может оказаться лучше, чем каждый отдельный результат. Некоторые навигационные системы используют эту идею, порождая точную оценку положения на основе нескольких оценок. Мы исследуем идею комбинирования частичных описаний в главе 7.

1.4 Исследовательское поведение

Исследовательское поведение — один из самых мощных механизмов повышения надежности биологических систем[16]. Идея состоит в том, что желаемый результат достигается при помощи процесса порождения и фильтрации гипотез (см. рис. 1.2). Такая организация позволяет порождающему механизму (генератору) быть достаточно общим и работать независимо от проверяющего механизма (тестера), который принимает либо отвергает каждый отдельный порожденный результат.

Например, важным компонентом жесткого скелета, поддерживающего форму клетки, служит массив микротрубочек. Каждая трубочка состоит из белковых единиц, формирующих ее. В живой клетке трубочки постоянно создаются и разрушаются и растут во всех направлениях. Однако стабильны только те из них, которые встречаются с кинетохором или другим стабилизатором в мембране клетки, и таким образом поддерживается форма, определяемая позицией стабилизаторов [71]. Поэтому механизм роста и поддержания формы относительно независим от механизма, определяющего эту форму. Такой механизм отчасти определяет форму клеток сложного организма и существует почти у всех животных.

[15] Напротив, часто можно слышать аргументы против встраивания дублирования в техническую систему. Например, философия компьютерного языка Python утверждает: «Должен существовать один — и желательно только один — очевидный способ сделать это» [95].

[16] Это утверждение исследуется в книге Киршнера и Герхарта [70].

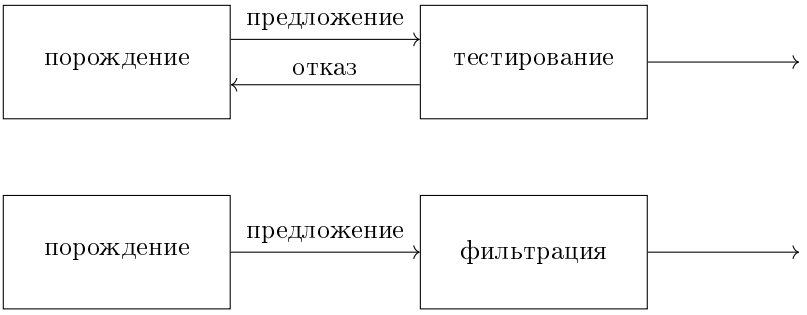

Рис. 1.2. Исследовательское поведение может быть организовано двумя способами. В одном случае порождающий механизм предлагает некоторое действие (или результат), а тестирующий компонент может явным образом его отвергнуть. В таком случае генератор должен породить альтернативу. Во втором случае генератор порождает все альтернативы без какой-либо обратной связи, а фильтр выбирает из них одну или более приемлемую

Исследовательское поведение наличествует на всех уровнях детализации биологических систем. Нервная система растущего эмбриона порождает значительно больше нейронов, чем будет нужно взрослому организму. Те из них, которые найдут подходящие цели в других нейронах, органах чувств или мышцах, выживут, а те, которые цели не найдут, самоуничтожатся. Рука образуется через рост пластинки с последующим уничтожением, методом апоптоза (программируемой смерти клеток), ненужного материала между пальцами [131]. Кости в нашем организме постоянно пересоздаются остеобластами (которые строят костный материал) и остеокластами (которые его разрушают). Форма и размер костей определяются ограничениями, задаваемыми их окружением: органами, с которыми им нужно связываться, а именно мышцами, связками, сухожилиями и другими костями.

Поскольку генератор не обязан знать, как тестер принимает или отвергает его предложения, а тестеру не обязательно знать, как формирует предложения генератор, эти два механизма могут развиваться независимо друг от друга. Эволюция и адаптация от этого становятся эффективнее, поскольку мутация в каждой из этих подсистем не обязана сопровождаться парной мутацией в другой. Однако, с другой стороны, такая изоляция может стоить дорого из-за цены, которую приходится платить за порождение и оценку неудачных предложений[17].

В сущности, порождение и проверка являются метафорой эволюции в целом. Механизмы биологического варьирования — случайные мутации, модификации генетических программ. Большинство этих мутаций нейтрально в том смысле, что они прямо не влияют на приспособленность по причине дублирования в системах. Естественный отбор служит фазой проверки. Он не зависит от того, как работает варьирование, а метод варьирования не знает заранее результатов отбора.

Существуют еще более удивительные явления: даже в близкородственных существах компоненты, которые выглядят почти одинаково во взрослом организме,

[17] Эти расходы можно значительно сократить, если имеется достаточная информация, позволяющая сократить на раннем этапе число подлежащих тестированию кандидатов. Мы рассмотрим изящный пример такой оптимизации в главе 7.

могут на стадии зародыша порождаться совершенно разным образом[18]. В случае дальнего родства различие механизмов построения сходных структур можно приписать «конвергентной эволюции», однако для близких родственников более разумно считать его свидетельством разделения уровней детализации, где результат определяется отчасти независимо от способа его достижения.

Инженерные системы могут иметь подобную же структуру. Мы стараемся отделять спецификацию от реализации: часто есть множество способов удовлетворить спецификацию, и проекты могут выбирать различные реализации. Лучший способ отсортировать набор данных зависит от размера этого набора и от вычислительной стоимости сравнения элементов. Наилучший метод представления многочлена зависит от того, плотный он или разреженный. Однако, даже если подобный выбор производится динамически (что само по себе необычно), он остается детерминистским: мы почти не знаем систем, одновременно применяющих несколько способов решения задачи и использующих тот ответ, который получен первым (зачем нам вообще все эти процессорные ядра?). Редки даже системы, пробующие несколько методов последовательно: если один потерпит неудачу, можно попробовать другой. Мы рассмотрим, как перебор с возвратами используется для реализации механизма генерации и проверок при сопоставлении с образцом, в главе 4. Мы научимся встраивать автоматический перебор с возвратами в язык в главе 5. Наконец, мы научимся строить механизм перебора, управляемого зависимостями, который извлекает как можно больше информации из неудач, в главе 7.

1.5 Цена гибкости

> Программисты на Lisp знают
> значение чего угодно, но
> ничему не знают цену.
>
> Алан Перлис, перефразируя
> Оскара Уайльда

Мы заметили, что в системах, использующих полиморфизм, многослойность, дублирование и исследовательское поведение, повышается общность и способность к развитию. Каждое из этих свойств само по себе достаточно дорого. Механизм, способный обрабатывать множество различных видов входных данных, для получения того же результата должен проделать больше работы, чем механизм, приспособленный к какому-то одному виду. Механизм с избыточностью содержит больше частей, чем эквивалентный ему механизм без избыточности. Механизм с дублированием кажется еще более расточительным. А механизм, демонстрирующий исследовательское поведение методом порождения и фильтрации результатов, легко может утонуть в экспоненциальном поиске. Однако в системах, способных развиваться, все это — ключевые способы решения задач. Возможно, при создании действительно

[18] Роговица курицы и роговица мыши почти идентичны, однако их морфогенез совершенно не совпадает; даже порядок формообразующих событий различается. Бард [7], раздел 3.6.1, сообщает, что наличие у разных видов различных методов формирования сходных структур — распространенное явление. Он приводит ряд примеров. Вот один особо впечатляющий: у лягушек *Gastrotheca riobambae* (см. дель Пино и Элинсон [28]) обычная лягушачья морфология развивается из эмбрионального диска, в то время как у остальных лягушек она происходит из приблизительно сферического эмбриона.

устойчивых систем мы должны быть готовы платить за достаточно сложно устроенную и дорогую инфраструктуру.

Часть проблемы тут в том, что мы думаем о стоимости в неправильных терминах. Цена времени и пространства имеет значение, но наша интуиция по поводу того, откуда происходят эти расходы, несовершенна. Каждому инженеру известно, что оценка реальной производительности системы требует множества тщательных измерений, которые часто показывают, что основные расходы случаются в неожиданных местах. При росте сложности задача становится только труднее. Однако мы продолжаем настаивать на преждевременной оптимизации на всех уровнях нашей программы, не зная их реального значения.

Допустим, нам удалось отделить те части нашей программы, которые должны быть быстрыми, от тех, что должны быть умными. При такой политике вся цена общности и способности к развитию ограничивается теми модулями, которые должны быть умными. В компьютерных системах такой взгляд необычен, однако в обычной жизни он встречается на каждом шагу. Когда мы пытаемся научиться чему-то новому, скажем игре на музыкальном инструменте, на ранних стадиях требуются сознательные действия, чтобы связать желаемый эффект с физическими движениями, вызывающими его. Но, по мере того, как наше умение возрастает, большая часть работы уже не требует сознательного внимания. Это обязательно должно произойти, чтобы научиться играть быстро, поскольку сознательные действия слишком медленные.

Похожее рассуждение проходит при разговоре о программном и аппаратном обеспечении. Аппаратура строится из соображений эффективности, и ценой этого является фиксированный интерфейс. На основе этого интерфейса можно написать программу — в сущности, создавая виртуальную машину. У этого дополнительного уровня абстракции есть известная цена, однако такой компромисс оправдан приобретаемой общностью. (Иначе мы бы до сих пор писали на ассемблере!) Мысль здесь состоит в том, что уровневая структура позволяет одновременно достичь и эффективности, и гибкости. Мы считаем, что требование реализовывать всю систему максимально эффективным способом контрпродуктивно, поскольку вредит гибкости, которая нужна для приспособления системы под будущие нужды.

Основная стоимость системы состоит во времени, затраченном программистами на проектирование, понимание, поддержку, модификацию и отладку системы. Поэтому ценность повышенной приспособляемости может быть даже еще больше. Легко изменяемая и приспособляемая система уничтожает один из главных компонентов стоимости: обучение новых программистов работе существующей системы во всех неприглядных деталях, чтобы они знали, куда полезть и где поправить код. В действительности цена нашей хрупкой инфраструктуры, возможно, намного превышает стоимость гибкого проектирования как из-за стоимости катастроф, так и из-за потерянных возможностей от чрезмерного времени, требуемого на перепроектирование и повторную реализацию. А если существенная доля времени, затраченного на переписывание системы под новые требования, заменяется приспособлением старой системы к новой ситуации, выигрыш еще больше.

Проблемы с корректностью

> Для оптимиста стакан наполовину полный. Для пессимиста стакан наполовину пустой. Для инженера стакан вдвое больше, чем надо.
>
> Автор неизвестен

Однако если мы строим системы так, что они применимы в большем числе ситуаций, чем те, что мы рассматриваем во время проектирования, возникает еще большая компонента стоимости. Поскольку мы оказываемся готовы использовать свои системы в контекстах, для которых они не были предназначены, нам неоткуда знать, что они при этом будут работать правильно!

Нас учат в информатике, что «корректность» программного обеспечения превыше всего и что она достигается путем построения формальных спецификаций подсистем и систем из этих подсистем и при помощи доказательства, что спецификации сочетания удовлетворяются на основе спецификаций отдельных подсистем и способа их сочетания[19]. Мы утверждаем, что такая дисциплина делает системы более хрупкими. Более того, для создания действительно устойчивых систем с этой жесткой дисциплиной надо распрощаться.

Проблема с требованием доказательств состоит в том, что, как правило, общие свойства гибких механизмов доказать сложнее, чем конкретные свойства конкретных механизмов в строго определенных условиях. Это побуждает нас как можно сильнее ограничивать спецификации и модули для того, чтобы упростить доказательства. Однако сочетание узкоспециализированных модулей оказывается хрупким — места для изменений не остается[20]!

Мы не собираемся агитировать против доказательств. Когда они есть, это замечательно. Без них нельзя обойтись в критических компонентах систем вроде сборщиков мусора (или рибосом)[21]. Однако даже для критично важных систем безопасности (вроде автопилотов) ограничение применимости ситуациями, когда соответствие поведения спецификации доказуемо, может привести к излишним поломкам в реальной жизни. Нам хотелось бы, чтобы автопилот пытался безопасно вести поврежденный самолет, даже если способ его повреждения не был предусмотрен проектировщиком!

Мы протестуем против дисциплины, когда доказательства *обязательны*: требование, чтобы применимость всего на свете была доказана для конкретной ситуации прежде, чем его можно было в этой ситуации применять, чрезмерно ограничива-

[19] Сложную систему трудно, если не невозможно, специфицировать. Как мы заметили на стр. 22, легко сформулировать требование, чтобы шахматная программа соблюдала правила шахмат, но как мы потребуем, чтобы она играла хорошо? А в отличие от шахмат, где хотя бы правила фиксированы, спецификации большинства систем постоянно изменяются по мере того, как меняются условия их использования. Как составить точное описание бухгалтерской системы в условиях постоянно перерабатываемых налоговых законов?

[20] В сущности, закон Постела (стр. 19) прямо противоречит практике построения систем из точно и узкоспецифицированных частей. Закон Постела рекомендует делать каждую компоненту более широко применимой, чем строго необходимо для каждого конкретного случая.

[21] Малозаметная ошибка в низкоуровневой подсистеме управления памятью, например в сборщике мусора, очень плохо поддается отладке — особенно если в системе есть параллельные процессы! Однако если ограничить сложность и размер таких подсистем, задача их спецификации и даже доказательства «правильности» становится выполнимой.

ет использование методов, повышающих устойчивость проектов. Это в особенности верно для методов, которые позволяют, с осторожностью, использовать какой-то прием за пределами его доказанной области применимости, и методов, которые предусматривают будущие расширения, не указывая заранее этому расширению четких границ.

К сожалению, многие из рекомендуемых нами методов значительно усложняют задачу доказательства, если не делают его практически невозможным. С другой стороны, иногда лучший способ решить задачу — обобщить ее до такой степени, что возникает простое доказательство.