Искусство программирования для UNIX

Эрик С. Реймонд

Серия "Программирование для профессионалов" от Addison-Wesley

Искусство программирования для Unix

Вдохновившим меня Кену Томпсону и Деннису Ритчи

Предисловие

Unix — не столько операционная система, сколько история. —Нил Стефенсон (Neal Stephenson)

Существует огромная разница между знаниями и опытом. Знания позволяют определить необходимые действия аналитическим путем, тогда как опыт делает верные действия рефлекторными, едва ли требующими осознанных размышлений вообще.

Целью книги является попытка преподнести читателям аспекты разработки программ в Unix, которые интуитивно известны экспертам данной операционной системы. Поэтому в данной книге, в отличие от большинства других книг о Unix, рассматривается меньше технических подробностей и больше вопросов

коллективной культуры , как в явной, так и в скрытой ее формах, а также ее осознанные и неосознанные традиции. Данная книга не дает ответов на вопросы о том, "как сделать что-либо", она не является сборником документов how-to, скорее в ней собраны ответы на вопросы наподобие "почему это следует сделать" (why-to).

Подход why-to обладает большой практической важностью, поскольку слишком многие программы проектируются неудачно. Для большинства из них характерен большой размер, чрезвычайная сложность сопровождения и чрезмерные трудности при переносе на новые платформы или расширении, которое не было предусмотрено создавшими их программистами. Данные проблемы являются симптомами ошибочного проектирования. Авторы надеются, что эта книга позволит популяризировать некоторые относящиеся к Unix знания в области хорошего проектирования.

Книга разделена на четыре части: "Контекст", "Проектирование", "Реализация" и "Сообщество". В первой части ("Контекст") освещены философские и исторические аспекты, способствующие созданию основы и заинтересованности для восприятия последующего материала. Во второй части ("Проектирование") принципы философии Unix разворачиваются в более специфические рекомендации, касающиеся проектирования и реализации. В третьей части ("Реализация") основное внимание уделено программному обеспечению Unix. Четвертая часть ("Сообщество") касается межличностного взаимодействия и соглашений, которые делают Unix-культуру столь эффективной в своей области.

Поскольку данная книга посвящена коллективной культуре, автор с самого начала не планировал создавать ее в одиночестве. Читатели заметят, что текст включает в себя высказывания выдающихся Unix-разработчиков, основателей традиций Unix. Эти светила были приглашены прокомментировать и обсудить текст в ходе длительного процесса публичного рассмотрения книги.

Используя слово "мы", автор не пытается показаться всеведущим, а старается отразить тот факт, что в книге предпринята попытка ясного выражения опыта всего сообщества.

Поскольку данная книга нацелена на передачу культуры, она включает в себя гораздо больше исторических фактов, фольклора и замечаний, чем обычная техническая книга. Наслаждайтесь: все это также является частью обучения Unix-программиста. Ни одна из исторических подробностей не является жизненно важной, но в совокупности все они существенны. Авторы полагают, что это делает книгу более интересной. Гораздо важнее понимание того, откуда произошла операционная система Unix и как она встала на путь своего развития. Это поможет развить интуитивное чувство Unix-стиля.

По той же причине авторы отказались от написания книги таким образом, как если бы история была завершенной. Читатели найдут большое количество ссылок относительно времени написания книги. Эти ссылки призваны обратить внимание читателя на то, что соответствующие факты могли устареть и должны быть перепроверены.

Кроме того, настоящая книга не является ни сборником уроков по языку C, ни руководством по командам и API-функциям операционной системы Unix. Она также не является справочником по программам

sed или

yacc , языкам Perl или Python, букварем для сетевого программиста или исчерпывающим описанием секретов системы X. Это не экскурс во внутреннее устройство или структуру операционной системы Unix. Указанные вопросы лучше рассмотрены в других книгах, и в тексте данной книги в соответствующих случаях имеются ссылки на них.

За пределами всех специфических технических вопросов Unix-культура обладает неписаной традицией, которая развивалась на протяжении миллионов человеко-лет[1] инженерной практики. Данная книга написана с верой в то, что понимание этой традиции и включение в свой инструментарий ее моделей поможет читателям стать лучшими программистами и проектировщиками.

Культура создается людьми, и традиционный путь изучения Unix-культуры состоит в постепенном ее восприятии через фольклор от членов сообщества. Эта книга не заменит передачи культуры в личном общении, но она способна ускорить этот процесс, позволяя изучить опыт других. Для кого предназначена эта книга

Книга рекомендуется для опытных Unix-программистов, которые обучают начинающих разработчиков либо спорят с приверженцами других операционных систем и находят трудным ясное изложение преимуществ Unix-подхода.

Книгу стоит прочесть программистам на C, C++ или Java, имеющим опыт работы в других операционных системах и планирующим начать Unix-проект.

Книгу также следует прочесть пользователям Unix, как новичкам, так и пользователям со средним уровнем квалификации, но имеющим небольшой опыт разработки и желающим изучить методики проектирования эффективного программного обеспечения в этой операционной системе.

Материал книги также рекомендуется изучить тем, кто, не являясь Unix-программистом, осознает ценность Unix-традиций. Авторы верят в правоту этой точки зрения; Unix-философию можно использовать в других операционных системах. Поэтому в данной книге этим другим системам (в особенности системам, разработанным корпорацией Microsoft) уделено больше внимания, чем обычно в книгах по Unix. Если инструментарий и учебные примеры применимы к таким системам, в тексте имеются соответствующие замечания.

Данная книга рекомендуется для изучения разработчикам прикладных программ, рассматривающим платформы или методы реализации для крупного общецелевого или вертикального приложения. Таким специалистам книга поможет понять преимущества Unix в качестве платформы для разработки, а также Unix-традиций применительно к открытым исходным кодам как методу разработки.

В то же время читателям

не стоит искать здесь подробности программирования на языке C или использования API-интерфейса ядра Unix. По этим темам имеется множество хороших книг, в частности:

"Advanced Programming in the Unix Environment" [81] — классический труд по изучению Unix API, а также книга

"The Practice of Programming" [40], которая входит в перечень рекомендованной литературы для всех программистов на языке C (а в действительности рекомендуется всем программистам на всех языках).Как использовать эту книгу

Данная книга одновременно является практической и философской. Некоторые ее части являются афористичными и общими, в других изучаются специфические примеры Unix-разработок. Рассмотрение общих принципов и афоризмов предваряется или завершается иллюстрирующими их примерами. Примеры взяты не из учебных программ, а из реально действующего кода, который используется ежедневно.

Авторы преднамеренно избегали наполнения книги длинными листингами или примерами файлов спецификаций, даже если во многих случаях это облегчало написание (а в некоторых, возможно, и чтение). В большинстве книг по программированию дано чрезмерное количество низкоуровневых деталей и примеров. Вместе с тем в них наблюдается недостаток высокоуровневого объяснения того, что в действительности происходит. Авторы данной книги предпочитают "заблуждаться в противоположном направлении".

Таким образом, несмотря на то, что читателям часто предлагается рассмотреть код или файлы спецификации, фактически в книгу включено сравнительно небольшое число таких примеров. Вместо них указаны ссылки на примеры, представленные на Web-ресурсах.

Усвоение данных примеров поможет превратить изученные принципы в полуинстинктивные практические знания. В идеальном случае данную книгу следует читать рядом с консолью работающей Unix-системы, имеющей удобный Web-браузер. Подойдет любая Unix-система, но уже установленные и немедленно доступные для изучения учебных примеров программы, вероятнее всего, будут находиться на какой-либо Linux-системе. Указатели в данной книге побуждают просматривать соответствующие страницы и экспериментировать.

Следует заметить: несмотря на все усилия, направленные на включение URL-адресов, которые должны оставаться стабильными и доступными, авторы не могут гарантировать их постоянную доступность. Если выяснится, что какая-либо процитированная ссылка устарела, рекомендуется, учитывая общий смысл, воспользоваться поиском словосочетаний в привычной поисковой машине в Web. Там, где это возможно, рядом с URL-адресами предложены способы поиска информации.

Большинство аббревиатур, использованных в данной книге, расшифровываются при первом использовании. Для удобства в приложении также приводится глоссарий.

Ссылки на дополнительную литературу обычно выполняются по номеру книги в списке (см. приложение B). Также даны нумерованные сноски на URL-адреса, которые могут затруднять чтение или предположительно являются ненадежными. Это же относится к примечаниям, историческим фактам и шуткам.

Для того чтобы сделать данную книгу более доступной для технически менее подготовленных читателей, она была предложена непрограммистам для прочтения и определения терминов, которые кажутся непонятными и одновременно необходимы для последовательного изложения. Также использовались сноски для определений элементарных понятий, которые вряд ли понадобятся опытным программистам. Дополнительные источники информации

Конечно, тематика данной книги уже рассматривалась в некоторых периодических изданиях и нескольких книгах, написанных первыми разработчиками операционной системы Unix. Среди них выделяется и по праву считается классической книга

"The Unix Programming Environment" [39] Кернигана (Kernighan) и Пайка (Pike). Однако в ней не рассматривается Internet и World Wide Web или новая волна интерпретируемых языков программирования, таких как Perl, Tcl и Python.

Работая над этой книгой, авторы внимательно изучали работу

"The Unix Philosophy" [26] Майка Ганкарза (Mike Gancarz). Данная книга является выдающейся в своем роде, однако Ганкарз не пытается раскрыть полный спектр тем, которые следовало бы рассмотреть. Тем не менее, авторы выражают благодарность ее создателю за напоминание о том, что простейшие модели проектирования в Unix являются наиболее устойчивыми и удачными.

Книга

"The Pragmatic Programmer" [37] отражает остроумную дискуссию о хорошей практике проектирования, которая относится к несколько другому, по сравнению с данной книгой, уровню искусства проектирования программ (в ней более подробно рассматриваются вопросы кодирования и меньше проблемы более высокого уровня). Философия ее автора базируется на опыте работы с Unix, а книга представляется отличным дополнением к данному изданию.

В книге

"The Practice of Programming" [40] рассматривается несколько тех же положений, что и в книге

"The Pragmatic Programmer" , но в аспекте Unix-традиции.

Наконец, авторы рекомендуют

"Zen Flesh, Zen Bones" [68], важную коллекцию основных источников Дзэн-буддиста. Ссылки на Дзэн включены в данную книгу, поскольку Дзэн предоставляет словарный запас для обозначения некоторых идей, которые оказываются весьма важными при проектировании программного обеспечения, но очень трудны для запоминания.Соглашения, используемые в данной книге

Термин "UNIX" технически и юридически является торговой маркой организации "The Open Group", и формально он должен использоваться только относительно сертифицированных операционных систем, прошедших сложные тесты на соответствие стандартам "The Open Group". В данной книге термин "Unix" используется в более свободном, широко распространенном среди программистов смысле. Здесь термин "Unix" применяется для обозначения любой операционной системы (имеющей формальное название Unix или нет), которая либо происходит из кода, унаследованного от системы Unix компании Bell Labs, либо "написана в строгом приближении к потомкам этой системы". В частности, Linux (из которой взято большинство примеров книги), согласно данному определению, является Unix-системой.

В настоящей книге используются соглашения справочной системы Unix (manual page) для выделения средств Unix с последующим указанием в скобках номера раздела справочной системы. Обычно это делается при первом упоминании команды, если требуется показать, что она принадлежит Unix. Например, запись "munger(1)" следует читать как "программа munger, описанная в разделе 1 (пользовательские средства) справочной системы Unix, в случае если она присутствует в данной системе". Раздел 2 содержит справочные сведения о системных вызовах C, раздел 3 — о вызовах библиотеки C, раздел 5 посвящен форматам файлов и протоколам, в разделе 8 описаны средства системного администрирования. Другие разделы отличаются в зависимости от принадлежности к различным Unix-системам, но они не цитируются в данной книге. Для получения более подробных сведений следует в приглашении командной строки Unix ввести команду man 1 man (в более давних системах ветви System V Unix может потребоваться команда man -si man).

Иногда в тексте упоминается какое-либо приложение Unix (такое как

Emacs), название которого приводится без указания номера раздела справочной системы и начинается с прописной буквы. Таким образом указывается название, которое фактически представляет широко распространенное семейство Unix-программ с одинаковыми функциями, и в тексте описываются общие свойства всех программ данного семейства. Например, семейство

Emacs включает в себя программу

xemacs .

Далее в настоящей книге нередко встречается ссылка на методы "старой школы" и "новой школы". Подобно рэп-музыке, новая школа возникла в 90-х годах прошлого века. В данном контексте новая школа связана с появлением языков написания сценариев (scripting language), графических пользовательских интерфейсов (Graphical User Interfaces — GUI), Unix-систем с открытым исходным кодом и Web-среды. Упоминание старой школы относится к периоду до 1990 года (и особенно до 1985 года), когда повсеместно применялись дорогостоящие (совместно используемые) компьютеры, частные Unix-системы, сценарии командного интерпретатора и программы на языке C. Данные различия стоит подчеркнуть, поскольку более дешевые машины с меньшими ограничениями памяти внесли значительные изменения в стиль Unix-программирования. Учебные примеры

Многие книги по программированию основаны на "игрушечных" примерах, сконструированных специально для подтверждения точки зрения автора. В данном случае это не так. Учебные примеры, приведенные в данной книге, являются реальными, существующими блоками программного обеспечения, которое используется ежедневно. Приведем некоторые из главных примеров.

cdrtools/xcdroast

Данные отдельные проекты обычно используются вместе. Пакет cdrtools представляет собой набор CLI-инструментов для записи дисков CD-ROM. Информацию по данному пакету можно найти в Web с помощью поискового слова "cdrtools". Приложение xcdroast — GUI-интерфейс для пакета cdrtools. Сайт проекта xcdroast доступен по адресу http://www.xcdroast.org.

fetchmail

Программа

fetchmail получает почту с удаленных почтовых серверов с помощью почтовых протоколов POP3 или IMAP. См. домашнюю страницу fetchmail <http://www.catb.org/~esr/fetchmail> (поиск в Web с помощью ключевого слова "fetchmail").

GIMP

GIMP (GNU Image Manipulation Program — GNU-программа для работы с графическими изображениями) полнофункциональная программа для создания и обработки изображений, которая способна осуществлять сложное редактирование множества различных графических форматов. Исходные коды программы доступны на домашней странице GIMP <http://www.gimp.org/> (или поиск в Web по слову "GIMP").

mutt

Пользовательский почтовый агент

mutt является лучшим среди современных текстовых Unix-агентов электронной почты, который известен благодаря хорошей поддержке MIME-форматов (Multipurpose Internet Mail Extensions — многоцелевые расширения почтового стандарта в Internet) и использованию таких средств безопасности, как PGP (Pretty Good Privacy) и GPG (GNU Privacy Guard). Исходный код продукта и исполняемые двоичные файлы доступны на сайте проекта Mutt http://www.mutt.org/.

xmlto

Команда

xmlto преобразует DocBook-документы и другие XML-документы в различные форматы, включая HTML, текстовый формат и PostScript. Исходные коды и документация представлены на сайте проекта xmlto <http://www.cyberelk.net/tim/xmlto/>.

В целях сокращения кода, который необходимо прочесть пользователю для понимания примеров, авторы старались подбирать те из них, которые могут использоваться несколько раз, в идеальном случае иллюстрируя несколько различных принципов и практических рекомендаций по проектированию. По той же причине многие примеры взяты из проектов автора. Их не следует рассматривать как наилучшие из возможных примеров, просто автор находит их достаточно известными для использования во многих демонстрационных целях.
Авторские благодарности

Приглашенные помощники Кен Арнольд (Ken Arnold), Стивен М. Белловин (Steven M. Bellovin), Стюарт Фельдман (Stuart Feldman), Джим Геттис (Jim Gettys), Стив Джонсон (Steve Johnson), Брайан Керниган (Brian Kernighan), Дэвид Корн (David Korn), Майк Леск (Mike Lesk), Дуг Макилрой (Doug McIlroy), Маршал Кирк Маккьюзик (Marshall Kirk McKusick), Кит Паккард (Keith Packard), Генри Спенсер (Henry Spencer) и Кен Томпсон (Ken Thompson) внесли крупный вклад в создание данной книги. В частности, Дуг Макилрой в очередной раз продемонстрировал свою преданность идеям превосходного качества, которые он привнес в управление еще первой исследовательской группой Unix тридцать лет назад.

Особую благодарность автор выражает Робу Лэндли и своей жене Кэтрин Реймонд, которые строку за строкой редактировали черновик рукописи. Внимательные и тонкие комментарии Роба вдохновили меня на создание нескольких полных глав, кроме того, его замечания во многом определили нынешнюю организацию книги и круг рассмотренных в ней тем. Если бы он написал весь тот текст, который он заставил меня улучшить, то я должен был бы называть его соавтором. Кэти играла роль моей тестовой аудитории, представляя читателей, которые не знакомы с техникой программирования.

В данную книгу вошли полезные моменты, выявленные в ходе обсуждения с другими специалистами в течение пяти лет ее написания. Марк М. Миллер (Mark M. Miller) способствовал автору в освещении темы параллельных процессов. Джон Коуэн (John Cowan) дал несколько ценных рекомендаций, касающихся моделей проектирования интерфейсов и

создал черновики учебных примеров программы

wily и системы VM/CMS. Джеф Раскин (Jef Raskin) продемонстрировал происхождение правила наименьшей неожиданности. Группа системной архитектуры UIUC (UIUC System Architecture Group) также внесла свои полезные дополнения. Рецензия членов группы вдохновила автора на написание разделов

"Что в Unix делается неверно" и

"Гибкость на всех уровнях". Рассел Дж. Нельсон (Russell J. Nelson) дополнил материал по образованию цепей Бернштайна в главе 7. Джей Мэйнард (Jay Maynard) значительно помог в разработке учебных примеров MVS в главе 3. Лес Хаттон (Les Hatton) предоставил множество полезных комментариев, касающихся главы

"Языки программирования: C или не C?" и побудил автора к созданию раздела

"Инкапсуляция и оптимальный размер модуля" в главе 4. Дэвид A. Вилер (David A. Wheeler) внес множество проницательных критических замечаний и некоторые материалы по учебным примерам, особенно в части

"Проектирование". Расс Кокс (Russ Cox) помог развить обзор системы Plan 9. Деннис Ритчи (Dennis Ritchie) корректировал некоторые исторические моменты, касающиеся языка C.

В период публичного рассмотрения книги с января по июнь 2003 года сотни Unix-программистов (слишком много, чтобы перечислить здесь их имена), вносили свои советы и комментарии. Как всегда, процесс открытия равноправного обсуждения посредством Web одновременно является крайне трудным и чрезвычайно полезным. Конечно, как всегда, всю ответственность за возможные ошибки автор принимает на себя.

На стиль изложения и некоторые вопросы, рассмотренные в данной книге, оказала определенное влияние известная концепция о моделях проектирования. Действительно, автор размышлял над названием

"Модели проектирования в Unix" (Unix Design Patterns). Однако оно было отклонено, поскольку автор был не согласен с некоторыми безоговорочными догмами данной школы и не испытывал необходимости использовать весь его формальный аппарат или принимать культурный багаж. Тем не менее, работы[2] Кристофера Александера (Christopher Alexander) (особенно

"The Timeless Way of Building" и

"A Pattern Language") оказали свое влияние на авторский подход. Автор считает своим долгом выразить огромную благодарность Группе четырех (Gang of Four) и другим членам их школы за демонстрацию того, каким образом можно использовать работы Александра в отношении проектирования на высоком уровне, не оперируя полностью неясными и бесполезными общими фразами. Заинтересованным читателям в качестве введения в тему моделей проектирования рекомендуется изучить книгу

"Design Patterns: Elements of Reusable Object-Oriented Software" [24].

Название данной книги, несомненно, ассоциируется с

"Искусством программирования" Дональда Кнута (Donald Knuth). Несмотря на то, что Кнут не связан с традициями Unix, он оказывает влияние на всех нас.

Редакторы с глубоким видением текста и богатым воображением встречаются не так часто, как хотелось бы. Один из них — Марк Тауб (Mark Taub), который смог оценить достоинства

приостановленного проекта и деликатно подтолкнул автора к окончанию работы. Хорошим чувством прозаического стиля и достаточными способностями улучшить написанное отличается и Мэри Лау Нор (Mary Lou Nohr). Джерри Вотта Oerry Votta) уловил авторскую идею обложки и сделал ее лучше, чем можно было представить. Весь коллектив издательства Addison-Wesley заслуживает высокой оценки за осуществление редактирования и производственного процесса, а также за терпимость к причудам автора, касавшимся не только текста, но и внешнего дизайна книги, оформления и маркетинга.

Часть I

Контекст

1

Философские вопросы

Те, кто не понимает Unix, приговорены к ее созданию, несчастные. Подпись из сообщений группы новостей Usenet, ноябрь 1987 года —Генри Спенсер

1.1. Культура? Какая культура?

Это книга о программировании в операционной системе Unix, но в ней неоднократно затрагиваются такие понятия, как "культура", "искусство" и "философия". Читателю, который не является программистом, или программисту, мало связанному с миром Unix, это может показаться странным. Однако операционная система Unix обладает собственной культурой; ей присуще особое искусство программирования и особая философия проектирования. Понимание этих традиций позволит разработчику создавать лучшее программное обеспечение, даже если оно не предназначено для Unix-платформ.

Каждой отрасли техники и проектирования присуща своеобразная техническая культура. В большинстве отраслей техники неписаные традиции являются частью образования практикующего специалиста, которая столь же важна, как официальные учебники и справочные руководства (а по мере накопления опыта часто является даже более важной). Старшие инженеры обнаруживают колоссальные объемы скрытых знаний, которые передаются их ученикам "особым путем" (как у Дзэн-буддистов), т.е. знания "распространяются посредством особой передачи вне священного писания".

Разработка программного обеспечения в общем случае является исключением из данного правила. Технология изменяется столь стремительно, программные среды появляются и исчезают настолько быстро, что т.н. техническая культура определяется как кратковременная и неустойчивая. В то же время это исключение также не всегда справедливо. Очень немногие программные технологии подтвердили свою долговечность, достаточную для развития устойчивой технической культуры, особого искусства и связанной с ним философии

проектирования, которые передаются от поколения к поколению инженеров.

Одним из примеров такой культуры является культура операционной системы Unix. Другим примером является культура Internet. Однако в двадцать первом веке можно утверждать, что обе эти культуры представляют собой единое целое. Обе они сформировались, и с начала 80-х годов прошлого века разделять их становится все труднее, поэтому в данной книге четкие границы между ними не проводятся.

1.2. Долговечность Unix

Операционная система Unix родилась в 1969 году и с момента возникновения находится в процессе постоянного использования и развития. Unix пережила несколько эпох, ограниченных стандартами компьютерной индустрии, — она старше, чем персональные компьютеры, рабочие станции, микропроцессоры или даже терминалы с видеодисплеями, и является современником первых полупроводниковых модулей памяти. Из всех современных систем разделения времени (timesharing systems) только о VM/CMS производства корпорации IBM можно утверждать, что она существует более продолжительный период, однако Unix-машины обеспечили в сотни тысяч раз больше служебных часов. Действительно, Unix, вероятно, поддерживает больший объем компьютерных вычислений, чем все остальные системы разделения времени.

Unix нашла свое применение в более широком диапазоне машин, чем любая другая операционная система. От суперкомпьютеров, рабочих станций и серверов, персональных и мини-компьютеров до карманных компьютеров и встроенного сетевого оборудования, Unix поддерживала и поддерживает, вероятно, больше архитектур и более разнообразное аппаратное обеспечение, чем какие-либо три другие операционные системы вместе взятые.

Операционная система Unix поддерживает невероятно широкий диапазон использования. Ни одна другая операционная система не служит одновременно в качестве инструмента исследований, дружественной основы для узкоспециальных технических приложений, платформы для коммерческого программного обеспечения бизнес-процессов и жизненно важного компонента технологии Internet.

Сомнительные предсказания о том, что Unix иссякнет или будет вытеснена другими операционными системами, постоянно высказываются с момента ее возникновения. Но до сих пор Unix, воплощенная сегодня в Linux, BSD Solaris и MacOS X и около десятка других вариантов, выглядит сильнее, чем когда-либо.

Роберт Меткалф (Robert Metcalf), создатель Ethernet, говорит, что если кто-либо разработает технологию, заменяющую Ethernet, то она будет названа "Ethernet", поэтому Ethernet не умрет никогда[3]. Unix уже пережила несколько подобных трансформаций. Кен Томпсон

Как минимум одна из центральных технологий Unix — язык C — широко распространен за пределами данной операционной системы. Действительно, в наши дни трудно представить разработку программного обеспечения без C, повсеместно используемого в качестве общего языка системного программирования. В Unix также были представлены широко распространенное в наши дни древовидное пространство имен файлов с узлами каталогов и конвейеры (pipeline) для сообщения программ.

Долговечность и способность Unix адаптироваться поистине удивительны. Другие технологии появляются и исчезают, как бабочки-однодневки. Мощность машин выросла в тысячи раз, языки трансформировались, промышленная практика пережила множество революций, а Unix

остается, продолжает функционировать, приносить доход и пользуется приверженностью со стороны множества лучших и талантливейших разработчиков программных технологий планеты.

Одним из многих последствий экспоненциального роста соотношения мощности и времени в вычислительной технике, а также огромных темпов разработки программного обеспечения является то, что знания специалиста наполовину устаревают каждые 18 месяцев. Операционная система Unix не устраняет данный феномен, но серьезно его сдерживает. Такие неизменные базовые элементы, как языки, системные вызовы и вызовы инструментальных средств, действительно можно использовать в течение многих лет и даже десятилетий. Для других же систем невозможно предсказать, что будет оставаться стабильным, поскольку даже целые операционные системы периодически выходят из употребления. В Unix прослеживается четкое отличие между временными и постоянными знаниями, и специалист может заранее узнать (с 90-процентной уверенностью), какая категория предмета вероятнее всего устареет в процессе его изучения. Такова лояльность Unix.

Во многом стабильность и успех рассматриваемой операционной системы связаны с конструкторскими решениями Кена Томпсона, Денниса Ритчи, Брайана Кернигана, Дуга Макилроя, Роба Пайка и других разработчиков первых версий Unix. Однако стабильность и успех Unix также связаны с проектной философией, искусством программирования и технической культурой, которые развивались вокруг Unix.

1.3. Доводы против изучения культуры Unix

Долговечность Unix и ее техническая культура, несомненно, интересны тому, кто уже является приверженцем данной операционной системы, и, возможно, историкам, изучающим развитие технологии. Однако первоначальное применение Unix в качестве общецелевой системы разделения времени для средних и крупных компьютеров быстро теряет свою актуальность благодаря появлению и развитию персональных рабочих станций. Есть определенные сомнения в том, что Unix когда-либо достигнет успеха на современном рынке настольных бизнес-приложений, где в настоящее время доминирует корпорация Microsoft.

Непрофессионалы часто отвергают Unix, считая ее академической игрушкой или "песочницей для хакеров". Так, широко известный спор,

Unix Hater's Handbook [27], длится почти столько же лет, сколько лет самой Unix, причем в ходе этого спора ее приверженцев слишком часто называли чудаками и неудачниками. Свою роль в этом, несомненно, сыграли колоссальные и повторяющиеся ошибки корпораций AT&T, Sun, Novell, а также других коммерческих поставщиков и консорциумов по стандартизации в позиционировании и маркетинговой поддержке Unix.

Даже "изнутри" данная система не выглядит стабильной и универсальной. Скептики говорят, что Unix слишком полезна, чтобы умереть, но слишком неудобна, чтобы вырваться из лабораторий, т.е. считают ее только "нишевой" операционной системой.

Более всего доводы скептиков опровергает подъем операционной системы Linux и других Unix-систем с открытым исходным кодом (таких как современные варианты BSD). Культура Unix показала себя "слишком живучей", чтобы разрушиться даже после десятка ошибок поставщиков. В настоящее время Unix-сообщество, принявшее на себя управление технологией и маркетингом, быстро и очевидно решает проблемы данной операционной системы (способы разрешения проблем более подробно рассматриваются в главе 20).

1.4. Что в Unix делается неверно

Для конструкции, начало которой было положено в 1969 году, в высшей степени трудно идентифицировать конструкторские решения, которые определенно являются ошибочными.

Так, Unix-файлы не имеют структур выше байтового уровня. Удаление файлов является необратимой операцией. Есть основания утверждать, что модель безопасности в Unix слишком примитивна. Управление задачами реализовано некачественно. Существует слишком много различных названий одних и тех же явлений. Целесообразность файловой системы вообще ставится под сомнение. Перечисленные технические проблемы рассматриваются в главе 20.

Однако, возможно, что наиболее веские возражения против Unix являются следствием одного из аспектов ее философии, впервые в явном виде выдвинутого разработчиками системы X Window. Система X стремится обеспечить "механизм, а не политику" ("mechanism, not policy"), поддерживая чрезвычайно общий набор графических операций и передвигая возможность выбора инструментального набора, а также внешнего вида интерфейса (то есть политику), на уровень приложения. Подобные тенденции характерны и для других служб системного уровня в Unix. Окончательный выбор режима работы все в большей степени определяется пользователем, для которого доступно целое множество оболочек (shells). Unix-программы обычно обеспечивают несколько вариантов работы и активно используют сложные средства представления.

Данная тенденция характеризует Unix как систему, разработанную главным образом для технических пользователей, и подтверждает, что пользователям известно о своих потребностях больше, чем разработчикам операционной системы.

Эта доктрина была четко определена в Bell Labs Диком Хеммингом (Dick Hamming)[4]. В 50-х годах прошлого века, когда компьютеры были редкими и дорогими, он настаивал на том, что система общественных вычислительных центров (open-shop computing), где клиенты имеют возможность писать собственные программы, является крайне необходимой. Он считал, что: "лучше решить правильно выбранную проблему неверным путем, чем верным путем решить не ту проблему". Дуг Макилрой.

Однако в результате такого подхода ("механизм, а не политика") определился следующий постулат: если пользователь

может установить политику, он

вынужден ее устанавливать. Нетехнических пользователей "ошеломляет" изобилие параметров и стилей интерфейсов в Unix, из-за чего они предпочитают системы, в которых, по крайней мере, создана видимость простоты.

В ближайшей перспективе "политика невмешательства" Unix может привести к потере большого количества нетехнических пользователей. Однако в долгосрочной перспективе может оказаться, что эта "ошибка" создает важнейшее преимущество, поскольку время жизни политики, как правило, коротко, и оно значительно меньше времени жизни механизма. Сегодняшняя мода на интуитивные интерфейсы слишком часто становится завтрашней тупиковой ветвью эволюции (как эмоционально скажут пользователи устаревших инструментальных средств X). Оборотная сторона заключается в том, что философия "механизм, а не политика" может позволить Unix восстановить свою актуальность после того, как конкуренты, которые сильнее привязаны к одному набору политик или вариантов

интерфейсов, пропадут из вида[5].

1.5. Что в Unix делается верно

Недавний взрывной рост популярности операционной системы Linux и возрастающая важность Internet дают весомые причины полагать, что доводы скептиков неверны. Однако даже если скептическая оценка справедлива, Unix-культуру стоит изучать, поскольку существует ряд моментов, которые в Unix реализованы гораздо лучше, чем в конкурирующих операционных системах.

1.5.1. Программное обеспечение с открытым исходным кодом

Несмотря на то, что понятия "открытый исходный код" (open source) и "определение открытого исходного кода" (open source definition) были сформулированы в 1998 году, коллективная разработка свободно распространяемого исходного кода была ключевой особенностью культуры Unix с момента ее возникновения.

В течение первых десяти лет первоначальная версия Unix, разработанная корпорацией AT&T, и ее основной вариант Berkeley Unix обычно распространялись с исходным кодом, что способствовало возникновению большинства других полезных явлений, которые описываются ниже.

1.5.2. Кроссплатформенная переносимость и открытые стандарты

Unix остается единственной операционной системой, которая в гетерогенной среде компьютеров, поставщиков и специализированного аппаратного обеспечения способна представить связный и документированный программный интерфейс приложений (Application Programming Interface — API). Она является единственной операционной системой, которую можно масштабировать от встроенных микросхем и карманных компьютеров до настольных машин, серверов и всего спектра вычислительной техники, включая узкоспециальные вычислительные комплексы и серверы баз данных.

API-интерфейс Unix — ближайший элемент к независимому от аппаратного обеспечения стандарту для написания действительно совместимого программного обеспечения. Не случаен тот факт, что стандарт, первоначально названный институтом IEEE

стандартом переносимых операционных систем (Portable Operating System Standard) , вскоре приобрел соответствующий суффикс и стал называться POSIX. Unix-эквивалент API был единственной заслуживающей доверия моделью для такого стандарта.

Приложения для других операционных систем, распространяемые в двоичном виде, исчезают вместе с породившими их средами, тогда как исходные коды Unix вечны, по крайней мере, в технической культуре Unix, которая совершенствует и поддерживает их в течение десятилетий.

1.5.3. Internet и World Wide Web

Контракт Министерства обороны США на первую реализацию набора протоколов TCP/IP был направлен группе разработчиков Unix, поскольку исходные коды данной операционной системы были в значительной степени открытыми. Кроме TCP/IP, Unix стала одной из необходимых центральных технологий индустрии ISP (Internet Service Provider — провайдер Internet-услуг). Со времени выхода из употребления семейства операционных систем TOPS в середине 80-х годов двадцатого века большинство Internet-серверов (а фактически все машины выше уровня персональных компьютеров) стали работать под управлением Unix.

Даже ошеломляющее маркетинговое влияние корпорации Microsoft не способно потеснить распространение операционной системы Unix в Internet. Хотя TCP/IP-стандарты (на которых основывается Internet) развивались в среде TOPS-10 и теоретически отделены от Unix, попытки заставить их работать в других операционных системах скованы несовместимостью, нестабильностью и ошибками. Теория и спецификации являются общедоступными, однако инженерные традиции, позволяющие преобразовать их в единую и работающую реальность, существуют только в мире Unix[6].

Слияние технической культуры Internet и Unix-культуры началось в начале 80-х годов прошлого века, и в настоящее время эти культуры нераздельно переплетены. Своей конструкцией технология World Wide Web, "современное лицо" Internet, настолько же обязана Unix, насколько и своей предшественнице — сети ARPANET. В частности, концепция универсального указателя ресурсов (Uniform Resource Locator — URL), центрального элемента Web является обобщением характерной для Unix идеи о едином пространстве именования файлов. Для того чтобы решать проблемы на уровне Internet-эксперта, понимание Unix и ее культуры является чрезвычайно важным.

1.5.4. Сообщество открытого исходного кода

Сообщество, первоначально сформированное вокруг ранних дистрибутивов Unix, уже никогда не исчезало. После бурного роста Internet в начале 90-х годов в его ряды было вовлечено все новое поколение увлеченных хакеров с домашними компьютерами.

В настоящее время это сообщество является мощной группой поддержки для всех видов разработки программного обеспечения. Мир Unix изобилует высококачественными средствами разработки с открытыми исходными кодами (многие из которых рассмотрены далее в настоящей книге). Unix-приложения с открытым исходным кодом обычно эквивалентны, а часто и превосходят свои частные аналоги [23]. Полные Unix-подобные операционные системы с совершенными инструментариями и пакетами основных приложений доступны для бесплатной загрузки через Internet. Зачем создавать программы с нуля, когда можно адаптировать, использовать повторно, перерабатывать и экономить 90% рабочего времени, используя общедоступный код?

Эта традиция совместного использования кода во многом зависит от тяжело дающегося опыта кооперативной разработки программ и их повторного использования, который накапливается не с помощью абстрактной теории, а в процессе длительной инженерной практики. Эти неочевидные правила разработки позволяют программам функционировать не только как изолированные однократно используемые решения, но и как синергетические

части инструментального набора. Главной целью данной книги является разъяснение этих правил.

В настоящее время "расцветающее" движение открытого исходного кода приносит в Unix-традиции новую жизненную силу, новые технические подходы и обогащает достижениями целых поколений талантливых молодых программистов. Проекты с использованием открытого исходного кода, включая операционную систему Linux и тесно связанные с ней компоненты, такие как Web-сервер Apache и Web-браузер Mozilla, придали Unix-традиции беспрецедентный уровень известности и успеха в современном мире. Движение открытого исходного кода, по всей видимости, выигрывает право на определение компьютерной инфраструктуры завтрашнего дня, и стержнем этой инфраструктуры станут Unix-машины, работающие в Internet.

1.5.5. Гибкость на всех уровнях

Многие операционные системы, называемые более "современными" или более "дружественными по отношению к пользователю", чем Unix, достигают "внешней красоты" путем связывания пользователей и разработчиков одной интерфейсной политикой. В таких системах предлагается программный интерфейс приложений, который при всей своей сложности является довольно ограниченным и жестким. Задачи, которые были предусмотрены проектировщиками, выполняются в них весьма просто, однако непредусмотренные задачи часто невозможно выполнить или их выполнение является крайне затруднительным.

С другой стороны, операционная система Unix обладает большой гибкостью. Множество способов, предусмотренных в Unix для соединения программ друг с другом, позволяет утверждать, что компоненты ее основного инструментального набора могут комбинироваться для создания полезных эффектов, которые разработчики отдельных частей этого набора не предусматривали.

Поддержка операционной системой Unix множества стилей программных интерфейсов (которая часто рассматривается как недостаток, поскольку увеличивает видимую сложность системы для конечных пользователей) также способствовала развитию гибкости; ни одна программа, которая должна быть просто блоком обработки данных, не несет на себе издержек, связанных со сложностью замысловатого графического пользовательского интерфейса (GUI).

В Unix-традиции значительный акцент сделан на сохранении сравнительно небольших размеров интерфейсов программирования, их чистоты и ортогональности, что также способствует гибкости систем. Несложные действия выполняются просто, а выполнение сложных действий, как минимум, возможно.

1.5.6. Особый интерес исследования Unix

Люди, провозглашающие техническое превосходство системы Unix, часто не уделяют достаточного внимания, может быть, наиболее важному ее преимуществу. Исследование Unix представляет особый интерес.

Кажется, что адептам Unix иногда бывает почти стыдно это признать. Как будто признание того, что они испытывают интерес, так или иначе может повредить их репутации. Тем не менее, это правда. Unix была и остается операционной системой, с которой интересно экспериментировать и в которой интересно разрабатывать программы.

Следует отметить, что это можно сказать далеко не о многих системах. Действительно, напряжение и труд разработчика в других средах часто сравнивают с выталкиванием мертвого кита с мели[7]. Напротив, в мире Unix операционная система скорее вознаграждает усилия, чем вызывает разочарование. Программисты в Unix обычно видят в ней не противника, победа над которым является главной целью, а скорее соратника.

В этом есть реальное экономическое значение. Фактор занимательности, конечно, сыграл свою роль. Людям нравилась Unix, поэтому они разрабатывали для нее больше программ, которые сделали ее использование приятным. В наши дни целые Unix-системы промышленного качества с открытым исходным кодом создаются людьми, которые воспринимают такую свою деятельность как хобби. Для того чтобы понять насколько это удивительно, достаточно попытаться найти людей, которые интереса ради занимаются клонированием операционных систем OS/360, VAX/VMS или Microsoft Windows.

Таким образом, фактор "интереса" ни в коем случае не является малозначительным. Те люди, которые становятся программистами или разработчиками, испытывают "интерес", когда их попытки, направленные на разрешение задачи, требуют от них изобретательности, но находятся в пределах их возможностей. Следовательно, "интерес" — это знак максимальной эффективности. Среды, в которых разработка программ становится тягостной, попусту "растрачивают" труд и творческое мышление, а впоследствии приводят к огромным скрытым затратам времени и средств.

Даже если бы Unix не характеризовалась иными достоинствами, то ее инженерную культуру стоило бы изучать для понимания тех ее особенностей, которые поддерживают интерес к разработке, поскольку именно этот фактор делает разработчика умелым и продуктивным.

1.5.7. Уроки Unix применимы в других операционных системах

Unix-программисты аккумулировали десятилетия опыта, пока наступил тот момент, когда функции новаторской операционной системы принимаются как должное. Даже программисты, не работающие с Unix, могут получить пользу от изучения данного опыта Unix. Так как Unix делает применение передовых принципов проектирования и методов разработки сравнительно простым, она является превосходной платформой для их изучения.

Другие операционные системы обычно делают хорошую практику более трудной, однако даже в этом случае можно использовать некоторые из уроков культуры Unix. Большая часть кода Unix (включая все ее фильтры, основные языки сценариев и многие генераторы кода) переносится непосредственно в любую операционную систему, поддерживающую ANSI C (по той причине, что язык C был "порождением" Unix, а библиотека ANSI C охватывает значительную часть Unix-служб).

1.6. Основы философии Unix

"Unix-философия" возникла вместе с ранними размышлениями Кена Томпсона о том, как сконструировать небольшую, но совершенную операционную систему с ясным служебным интерфейсом. Она развивалась по мере того, как Unix-культура изучала возможности получения максимального эффекта от проекта Томпсона, и в процессе этого развития впитывала уроки "предыдущей истории".

Философия Unix не является формальным методом проектирования. Она не была та свыше" как способ создания теоретически идеального программного обеспечения. Unix-философия (как удачные народные традиции в других инженерных дисциплинах) прагматична и основывается на опыте. Ее следует искать не в официальных методах и стандартах, а скорее в скрытых почти рефлекторных знаниях и

опыте ,

ко торый передается культурой Unix. Она культивирует чувство меры и достаточный скептицизм.

Дуг Макилрой, изобретатель каналов (pipes) в Unix и один из основателей Unix-традиции, сформулировал постулаты философии Unix следующим образом [52].

(i) Заставьте каждую программу хорошо выполнять одну функцию. Для решения новой задачи следует создавать новую программу, а не усложнять старые программы добавлением новых функций.

(ii) Будьте готовы к тому, что вывод каждой программы станет вводом другой, еще неизвестной программы. Не загромождайте вывод посторонней информацией. Избегайте строгих табличных или двоичных форматов ввода. Не настаивайте на интерактивном вводе.

(iii) Проектируйте и создавайте программное обеспечение, даже операционные системы, которые можно будет проверить в самом начале, в идеальном случае в течение нескольких недель. Без колебаний выбрасывайте громоздкие части и перестраивайте их.

(iv) Вместо неквалифицированной помощи используйте инструменты, облегчающие программирование, даже если инструменты приходится создавать "окольными" путями, зная, что некоторые из них будут удалены по окончании использования.

Позднее он обобщил эти положения (процитировано в книге

"A Quarter Century of Unix" [74]).

Вот в чем заключается философия Unix: пишите программы, которые выполняют одну функцию и делают это хорошо; пишите программы, которые будут работать вместе; пишите программы, поддерживающие текстовые потоки, поскольку они являются универсальным интерфейсом.

Роб Пайк, который стал одним из великих мастеров языка C, в книге

"Notes on C Programming" [62] предлагает несколько иной угол зрения.

Правило 1. Невозможно сказать, где возникнет задержка в программе. "Бутылочные горлышки" возникают в неожиданных местах, поэтому не следует пытаться делать предсказания и исправлять скорость до тех пор, пока не будет точно выяснено, где находится "бутылочное горлышко".

Правило 2. Проводите измерения. Не следует регулировать скорость до тех пор, пока не проведены измерения, и даже после измерений, если одна часть кода подавляет остальные.

Правило 3. Вычурные алгоритмы очень медленные, когда величина n является малой, а она обычно мала. Вычурные алгоритмы имеют большие константы. До тех пор, пока не известно, что n периодически стремится к большим значениям, не следует усложнять алгоритмы. (Даже если n действительно достигает больших значений, сначала используйте правило 2.)

Правило 4. Вычурные алгоритмы более склонны к появлению ошибок, чем простые, и их гораздо сложнее реализовать. Используйте простые алгоритмы, а также простые структуры данных.

Правило 5. Данные доминируют. Если выбраны правильные структуры данных и все организовано хорошо, то алгоритмы почти всегда будут очевидными. Для программирования центральными являются структуры данных, а не алгоритмы[8].

Правило 6. Правила 6 нет.

Кен Томпсон, спроектировавший и реализовавший первую Unix, усилил четвертое правило Пайка афористичным принципом, достойным Дзэн-патриарха:

В случае сомнений используйте грубую силу.

Гораздо сильнее Unix-философия была выражена не высказываниями старейшин, а их действиями, которые воплощает сама Unix. В целом, можно выделить ряд идей.

1. Правило модульности: следует писать простые части, связанные ясными интерфейсами.

2. Правило ясности: ясность лучше, чем мастерство.

3. Правило композиции: следует разрабатывать программы, которые будут взаимодействовать с другими программами.

4. Правило разделения: следует отделять политику от механизма и интерфейсы от основных модулей.

5. Правило простоты: необходимо проектировать простые программы и "добавлять сложность" только там, где это необходимо.

6. Правило расчетливости: пишите большие программы, только если после демонстрации становится ясно, что ничего другого не остается.

7. Правило прозрачности: для того чтобы упростить проверку и отладку программы, ее конструкция должна быть обозримой.

8. Правило устойчивости: устойчивость— следствие прозрачности и простоты.

9. Правило представления: знания следует оставлять в данных, чтобы логика программы могла быть примитивной и устойчивой.

10. Правило наименьшей неожиданности: при проектировании интерфейсов всегда следует использовать наименее неожиданные элементы.

11. Правило тишины: если программа не может "сказать" что-либо неожиданное, то ей вообще не следует "говорить".

12. Правило исправности: когда программа завершается аварийно, это должно происходить явно и по возможности быстро.

13. Правило экономии: время программиста стоит дорого; поэтому экономия его времени более приоритетна по сравнению с экономией машинного времени.

14. Правило генерации: избегайте кодирования вручную; если есть возможность, пишите программы для создания программ.

15. Правило оптимизации: создайте опытные образцы, заставьте их работать, прежде чем перейти к оптимизации.

16. Правило разнообразия: не следует доверять утверждениям о "единственно верном пути".

17. Правило расширяемости: проектируйте с учетом изменений в будущем, поскольку будущее придет скорее, чем кажется.

Новичкам в Unix стоит поразмышлять над данными принципами. Технические тексты по разработке программного обеспечения рекомендуют большую их часть, однако в других операционных системах, как правило, имеется недостаток необходимых средств и традиций для их внедрения, поэтому большинство программистов не могут применять их последовательно. Они принимают несовершенные инструменты, плохие конструкции, перенапряжение и распухший код как должное, а впоследствии удивляются, почему все это раздражает поклонников Unix.

1.6.1. Правило модульности: следует писать простые части, связанные ясными интерфейсами

Как однажды заметил Браян Керниган, "управление сложностью является сущностью компьютерного программирования" [41]. Отладка занимает большую часть времени разработки, и выпуск работающей системы обычно в меньшей степени является результатом талантливого проектирования, и в большей — результатом должного управления, исключающего многократное повторение ошибок.

Трансляторы, компиляторы, блок-схемы, процедурное программирование, структурное программирование, "искусственный интеллект", языки четвертого поколения, объектно-ориентированные языки и бесчисленные методологии разработки программного обеспечения рекламировались и продавались как средство борьбы с этой проблемой. Все они потерпели неудачу, если только их успех заключался в повышении обычного уровня сложности программы до той точки, где (вновь) человеческий мозг едва ли мог справиться. Как заметил Фред Брукс [8], "серебряной пули не существует".

Единственным способом создания сложной программы, не обреченной заранее на провал, является сдерживание ее глобальной сложности, т.е. построение программы из простых частей, соединенных четко определенными интерфейсами, так что большинство проблем являются локальными, и можно рассчитывать на обновление одной из частей без разрушения целого.

1.6.2. Правило ясности: ясность лучше, чем мастерство

Поскольку обслуживание является важным и дорогостоящим, следует писать такие программы, как если бы обмен наиболее важной информацией, осуществляемый программой, был связан не с компьютером, выполняющим данную программу, а с людьми, которые будут читать и поддерживать исходный код в будущем (включая и самого создателя программы).

В традициях Unix смысл данной рекомендации выходит за пределы простого комментирования кода. Хорошая Unix-практика также предполагает выбор алгоритмов и реализации с учетом дальнейшего обслуживания. Незначительный рост производительности ценой большого повышения сложности и запутанности методики относится к плохой практике не только потому, что сложный код, вероятнее всего, скрывает в себе ошибки, но также потому, что такой код будет тяжелее читать будущим кураторам (maintainers) программы.

С другой стороны, тот, кому впоследствии придется изменять программу, вряд ли будет поставлен в тупик изящным и ясным кодом — более вероятно, что он немедленно в нем разберется. Это особенно важно, если спустя несколько лет следующим куратором, возможно, будет сам создатель программы.

Не пытайтесь трижды расшифровывать хитроумный код. Однажды возможна неповторимая удача, однако, если придется разбираться в коде во второй раз, поскольку впервые он рассматривался слишком давно и детали позабыты, то это означает, что необходимо внести в код комментарии таким образом, чтобы третий раз был относительно безболезненным. Генри Спенсер.

1.6.3 Правило композиции: следует разрабатывать программы, которые будут взаимодействовать с другими программами

Если разрабатываемые программы не способны взаимодействовать друг с другом, то очень трудно избежать создания сложных монолитных программ.

Традиция Unix значительно способствует написанию программ, которые считывают и записывают простые текстовые форматы, ориентированные на потоки и не зависящие от устройств. В классической реализации Unix как можно больше программ создаются в виде простых

фильтров (filters) , которые на входе принимают простой текстовый поток и преобразовывают его в другой простой текстовый поток на выходе.

Вопреки распространенному мифу, такая практика широко распространена не потому, что Unix-программисты ненавидят графические пользовательские интерфейсы. Данная практика пользуется популярностью Unix-программистов, поскольку связывать вместе программы, не принимающие и не создающие на выходе простые текстовые потоки, гораздо труднее.

Текстовые потоки для Unix-инструментов являются тем же, чем являются сообщения для объектов в объектно-ориентированной среде. Простота интерфейса текстовых потоков усиливает инкапсуляцию инструментальных средств. Более сложные формы межпроцессного взаимодействия, такие как удаленный вызов процедур (remote procedure call), демонстрируют тенденцию к использованию программ с сильными внутренними зависимостями друг от друга.

Для того чтобы создавать компонуемые программы, их следует делать независимыми. Программа с одной стороны текстового потока как можно меньше должна зависеть от программы на другой стороне. Должна быть обеспечена возможность замены программы с одной стороны абсолютно другой реализацией без нарушения работы второй стороны.

GUI-интерфейсы могут быть разработаны на весьма высоком уровне. Иногда просто невозможно избежать сложных двоичных форматов данных какими-либо приемлемыми способами. Однако прежде чем создавать GUI-интерфейс, разумно будет изучить

возможность выделения частей разрабатываемой программы со сложным взаимодействием в один блок, а основных алгоритмов — в другой, а также использования простого потока команд или прикладного протокола для связи двух блоков. Прежде чем изобретать нетривиальный двоичный формат для распространения данных, стоит экспериментальным путем проверить, имеется ли возможность задействовать простой текстовый формат и согласиться с небольшими затратами на синтаксический анализ в обмен на возможность обработки потока данных с помощью универсальных инструментальных средств.

В ситуации, когда сериализованный интерфейс, подобный протоколу, не является естественным для разрабатываемого приложения, конструкция "в духе Unix" заключается в том, чтобы, по крайней мере, организовать как можно больше программных примитивов в библиотеку с хорошо определенным API-интерфейсом. В таком случае открываются возможности вызова приложений путем связывания или привязки к приложению нескольких интерфейсов для различных задач.

Данная проблема подробнее обсуждается в главе 7.

1.6.4. Правило разделения: следует отделять политику от механизма и интерфейсы от основных модулей

В разделе "Что в Unix делается неверно" отмечалось, что разработчики системы X Window приняли основное решение о реализации "механизма, а не политики". Такой подход был направлен на то, чтобы сделать систему X общим сервером обработки графики и передать решение о стиле пользовательского интерфейса инструментариям и другим уровням системы. Это оправданно, если учесть, что политика и механизм стремятся к изменению на протяжении различных периодов времени, причем политика меняется гораздо быстрее, чем механизм. Мода на вид и восприятие GUI-инструментариев может прийти и уйти, но растровые операции и компоновка останутся.

Таким образом, жесткое объединение политики и механизма имеет два отрицательных эффекта. Во-первых, политика становится негибкой и усложняется ее изменение в ответ на пользовательские требования. Во-вторых, это означает, что попытка изменения политики имеет строгую тенденцию к дестабилизации механизмов.

С другой стороны, разделение двух этих элементов делает возможным экспериментирование с новой политикой без разрушения механизмов. Кроме того, облегчается написание хороших тестов для механизма (политика, ввиду своего быстрого изменения, часто не оправдывает вложений).

Данное правило проектирования широко применимо вне контекста GUI-интерфейсов. В целом оно подразумевает, что следует искать способы разделения интерфейсов и основных модулей.

Одним из способов осуществления такого разделения, в частности, является написание приложения в виде библиотеки служебных подпрограмм на C, которые приводятся в действие встроенным языком сценариев, а управляющая логика приложения вместо C написана на языке сценариев. Классическим примером такой модели является редактор

Emacs, в котором для управления редактирующими примитивами, написанными на C, используется встроенный интерпретатор языка Lisp. Такой стиль программирования обсуждается в главе 11.

Другой способ заключается в разделении приложения на взаимодействующие процессы клиента (front-end) и сервера (back-end), которые обмениваются данными через специализированный протокол прикладного уровня посредством сокетов. Данный конструкторский подход рассматривается в главах 5 и 7. Во внешней или клиентской части реализуется политика, а во внутренней или серверной — механизм. Глобальная сложность данной пары часто является значительно меньшей, чем сложность одного процесса, в котором монолитно реализованы те же функции. Одновременно уменьшается уязвимость относительно ошибок и сокращаются затраты на жизненный цикл.

1.6.5. Правило простоты: необходимо проектировать простые программы и "добавлять сложность" только там, где это необходимо

Многие факторы приводят к усложнению программ (а следовательно, делают их более дорогими и более уязвимыми относительно ошибок). Программисты — это зачастую яркие люди, которые гордятся (часто заслуженно) своей способностью справляться со сложностями и ловко обращаться с абстракциями. Часто они состязаются друг с другом, пытаясь выяснить, кто может создать "самые замысловатые и красивые сложности". Столь же часто их способность проектировать превалирует над способностью реализовывать и отлаживать, а результатом является дорогостоящий провал.

Мнение о "замысловатой и красивой сложности" является почти оксюмороном. Unix-программисты соперничают друг с другом за "простоту и красоту". Эта мысль неявно присутствует в данных правилах, но ее стоит сделать очевидной. Дуг Макилрой.

Еще чаще (по крайней мере, в мире коммерческого программного обеспечения) излишняя сложность исходит от проектных требований, которые скорее основаны на маркетинговой причуде месяца, а не на реальных пожеланиях заказчика или фактических возможностях программы. Множество хороших конструкций были раздавлены маркетинговым нагромождением "статей контрольного перечня" — функций, которые часто вообще не востребованы заказчиком. Круг замыкается; соперники полагают, что должны соревноваться с чужими "украшательствами" путем добавления собственных. Довольно скоро "массивная опухоль" становится индустриальным стандартом, и все используют большие, переполненные ошибками программы, которые не способны удовлетворить даже их создателей.

В любом случае, в конечном результате проигрывают все.

Единственным способом избежать этих ловушек является поощрение культуры программного обеспечения, представители которой знают, что компактность красива, и активно сопротивляются "раздуванию" и сложности кода. Речь идет об инженерной традиции, высоко оценивающей простые решения и ищущей способы разделения программных систем на небольшие взаимодействующие блоки, традиции, которая борется с попытками создания программ с большим количеством "украшательств" (или даже хуже — проектирования программ

вокруг "украшательств").

Таковой была бы культура, во многом похожая на культуру Unix.

1.6.6 Правило расчетливости: пишите большие программы, только если после демонстрации

становится ясно, что ничего другого не остается

Под "большими программами" в здесь подразумеваются программы с большим объемом кода и значительной внутренней сложностью. Разрешая программе разрастаться, разработчик ставит под удар дальнейшее обслуживание данной программы. Поскольку люди неохотно расстаются с видимым результатом длительной работы, крупные программы привлекают излишние вложения в подходах, которые ошибочны или неоптимальны.

Проблема оптимального размера программного обеспечения более подробно рассмотрена в главе 13.

1.6.7. Правило прозрачности: для того чтобы упростить проверку и отладку программы, ее конструкция должна быть обозримой

Поскольку отладка часто занимает три четверти или более времени разработки, раннее окончание работы в целях облегчения отладки может быть очень хорошим решением. Особенно эффективный способ простой отладки заключается в проектировании с учетом

прозрачности (transparency) и

воспринимаемости (discoverability) .

Программная система

прозрачна, если при ее минимальном изучении можно немедленно понять, что она делает и каким образом. Программа

воспринимаема, когда она имеет средства для мониторинга и отображения внутреннего состояния, так что программа не только работает хорошо, но и позволяет

понять признаки правильной работы.

Проектирование с учетом данных характеристик имеет ряд последствий на всем протяжении проекта. Как минимум, это означает, что отладочные возможности не должны запаздывать. Скорее они должны быть спроектированы с самого начала — с той точки зрения, что программа должна быть способна как продемонстрировать собственную корректность, так и передать будущим разработчикам ментальную модель решаемой программой проблемы, разработанную создателем программы.

Для того чтобы программа могла продемонстрировать свою корректность, в ней должны использоваться достаточно простые форматы входных и выходных данных, с тем чтобы можно было легко проверить соответствующую связь между допустимым вводом и корректным выводом.

Цель разработки с учетом прозрачности и воспринимаемости также должна поддерживать простые интерфейсы, которыми могут легко манипулировать другие программы, в особенности средства тестирования и мониторинга, а также отладочные сценарии.

1.6.8. Правило устойчивости: устойчивость — следствие прозрачности и простоты

Программное обеспечение называют

устойчивым, когда оно выполняет свои функции в неожиданных условиях, которые выходят за рамки предположений разработчика, столь же хорошо, как и в нормальных условиях.

Большинство программ "являются хрупкими и переполненными ошибками, поскольку для человеческого мозга слишком сложно сразу понять всю программу целиком. Если нет возможности сделать корректный вывод о внутреннем устройстве программы, то нельзя быть уверенным в ее корректности, и ее невозможно исправить в случае поломки.

Следовательно, способ создания устойчивого программного обеспечения заключается в организации такого внутреннего устройства программ, которое было бы простым для понимания человеком. Существует два основных способа достижения этой цели: прозрачность и простота.

Для достижения устойчивости очень важным является проектирование, допускающее необычный или крайне объемный ввод. Этому способствует правило композиции; ввод, сгенерированный другими программами, известный для программ нагрузочных испытаний (например, оригинальный Unix-компилятор С по имеющимся сообщениям нуждался в небольшом обновлении, для того чтобы хорошо обрабатывать вывод утилиты Yacc). Используемые формы часто кажутся бесполезными для людей. Например, допускаются пустые списки, строки и т.д. даже в тех местах, где человек редко или никогда не вводит пустую строку. Это предотвращает особые ситуации при механическом генерировании ввода. Генри Спенсер.

Один весьма важный тактический прием для достижения устойчивости при работе с необычным вводом заключается в том, чтобы избегать частных случаев в коде. Часто ошибки скрываются в коде для обработки частных случаев, а также возникают в процессе взаимодействия частей кода, предназначенных для обработки различных частных случаев.

Как отмечалось выше, программа является

прозрачной в том случае, если при ее минимальном изучении немедленно становится ясно, что происходит. Программа является

простой, если происходящее в ней не представляется сложным для восприятия человеком. Чем сильнее эти свойства проявляются в разрабатываемых программах, тем устойчивее они будут.

Модульность (простые блоки, ясные интерфейсы) является способом организации программ, позволяющим сделать их проще. Существуют и другие способы достижения простоты. Ниже описывается один из них.

1.6.9. Правило представления: знания следует оставлять в данных, чтобы логика программы могла быть примитивной и устойчивой

Даже простейшую процедурную логику трудно проверить, однако весьма сложные структуры данных являются довольно простыми для моделирования и анализа. Для того чтобы убедиться в этом, достаточно сравнить выразительную и дидактическую силу диаграммы, например, дерева указателей, имеющего 50 узлов с блок- схемой программы, состоящей из 50 строк кода. Или сравнить инициализатор массива, выражающий таблицу преобразования,

с эквивалентным оператором выбора. Различие в прозрачности и ясности поразительно. См. правило 5 Роба Пайка.

Данные более "податливы", чем программная логика. Это означает, что если можно выбирать между сложностью структуры данных и сложностью кода, следует выбирать первое. Более того, развивая проект, следует активно искать пути перераспределения сложности из кода в данные.

Первые оценки этого факта не связаны с Unix-сообществом, но множество примеров Unix-кода отражают его влияние. В частности, средство языка C, манипулирующее указателями, способствовало использованию динамически модифицируемых ссылочных структур на всех уровнях кода, начиная с ядра. Простое отслеживание указателей в таких структурах часто выполняет такие задачи, которые в других языках потребовали бы внедрения более сложных процедур.

Данные методики также описываются в главе 9.

1.6.10. Правило наименьшей неожиданности: при проектировании интерфейсов всегда следует использовать наименее неожиданные элементы

Данное правило также широко известно под названием "Принцип наименьшего удивления" (Principle of Least Astonishment).

Простейшими в использовании являются программы, требующие наименьшего обучения пользователя, или, иными словами, простейшими в использовании программами являются программы, которые наиболее действенно ассоциируются с уже имеющимися у пользователя знаниями.

Таким образом, в дизайне интерфейса следует избегать беспричинной новизны и излишней "заумности". В программе-калькуляторе знак "+" всегда должен означать операцию сложения. Разрабатывая интерфейс, его следует моделировать с интерфейсов функционально подобных или аналогичных программ, с которыми пользователи вероятнее всего знакомы.

Необходимо учитывать характер предполагаемой аудитории. С программой могут работать конечные пользователи, другие программисты или системные администраторы.

Необходимо уделять внимание традиции. В мире Unix имеются довольно хорошо проработанные соглашения о таких элементах, как формат конфигурационных файлов, ключи командной строки и другие. Для существования этих традиций имеется весомая причина: они позволяют смягчить кривую освоения. Учитесь и используйте их.

Многие из этих традиций рассматриваются в главах 5 и 10.

Правило наименьшей неожиданности имеет оборотную сторону. Следует избегать выполнения внешне похожих вещей, слегка отличающихся в действительности. Это крайне опасно, поскольку кажущаяся привычность порождает ложные ожидания. Часто бывает лучше делать заметно отличающиеся вещи, чем делать их

почти одинаковыми.Генри Спенсер.

1.6.11. Правило тишины: если программа не может "сказать" что-либо неожиданное, то ей

вообще не следует "говорить"

Одно из старейших и наиболее постоянных правил проектирования в Unix гласит: если программа не может "сказать" что-либо интересное или необычное, то ей следует

"молчать" . Правильно организованные Unix-программы выполняют свою работу "ненавязчиво", с минимальным шумом и беспокойством. Молчание — золото.

Правило "молчание — золото" развивалось изначально, поскольку операционная система Unix предшествовала видеодисплеям. На медленных печатающих терминалах в 1969 году каждая строка излишнего вывода вызывала серьезные потери пользовательского времени. В наши дни данное ограничение снято, однако веские причины для краткости остаются.

Я думаю, что лаконичность Unix-программ является главной чертой стиля. Когда вывод одной программы становится вводом другой, идентификация необходимых фрагментов должна быть простой. Остается актуальным и человеческий фактор — важная информация не должна смешиваться с подробными сведениями о внутренней работе программы. Если вся отображаемая информация является важной, то найти важную информацию просто. Кен Арнольд.

Изящные программы трактуют внимание и сосредоточенность пользователя как ценный и ограниченный ресурс, который требуется только в случае необходимости.

Более подробно правило тишины и причины для его соблюдения описаны в конце главы 11.

1.6.12. Правило исправности: когда программа завершается аварийно, это должно происходить явно и по возможности быстро

Программное обеспечение должно быть столь же прозрачным при выходе из строя, как и при нормальной работе. Лучше всего, если программа способна справиться с неожиданными условиями путем адаптации к ним, однако наихудшими ошибками являются те, за которыми не следует восстановление, и проблема незаметно приводит к разрушению, проявляющемуся намного позднее.

Таким образом, следует писать программное обеспечение, которое как можно изящнее справляется с некорректным вводом и собственными ошибками выполнения. Однако если программа не способна справиться с ошибкой, то необходимо заставить ее прекратить выполнение таким способом, который на сколько это возможно упростит диагностику проблемы.

Рассмотрим также рекомендацию Постела (Postel)[9]: "Будьте либеральны к тому, что принимаете, и консервативны к тому, что отправляете". Постел говорил о программах сетевых служб, однако лежащая в основе такого подхода идея является более общей. Изящные программы сотрудничают с другими программами, извлекая как можно больше смысла из некорректно сформированных входных данных, и либо шумно прекращают свою работу, либо передают абсолютно четкие и корректные данные следующей программе в цепочке.

В то же время следует учитывать следующее предостережение.

Исходные HTML-документы рекомендовали "быть великодушными к тому, что принимаете", и

с тех пор это сбивало нас с толку, поскольку каждый браузер принимает другое подмножество спецификаций. Именно

спецификации должны быть "великодушны", а не их интерпретация.Дуг Макилрой.

Макилрой убеждает нас великодушно

проектировать, а не компенсировать неадекватные стандарты с помощью всепозволяющих реализаций. Иначе, как он правильно отмечает, очень просто все закончится смешением разметки.

1.6.13. Правило экономии: время программиста стоит дорого; поэтому экономия его времени более приоритетна по сравнению с экономией машинного времени

"В ранние мини-компьютерные времена Unix" вынесенная в заголовок идея была довольно радикальной (машины тогда работали намного медленнее и были более дорогими). В настоящее время, когда каждая группа разработчиков и большинство пользователей (за исключением нескольких лабораторий по моделированию ядерных взрывов или созданию 3D-анимации) обеспечены дешевыми машинными циклами, она может показаться слишком очевидной, чтобы о ней говорить.

Хотя почему-то практика, видимо, отстает от реальности. Если бы этот принцип принимался действительно серьезно в процессе разработки программного обеспечения, то большинство приложений были бы написаны на высокоуровневых языках типа Perl, Tcl, Python, Java, Lisp и даже на языках командных интерпретаторов — т.е. на языках, сокращающих нагрузку на программиста, осуществляя собственное управление памятью ([65]).

И это действительно происходит в мире Unix, хотя за его пределами большинство разработчиков приложений, кажется, не в состоянии отойти от стратегии старой школы Unix, предполагающей кодирование на C (или C++). Данная стратегия и связанные с ней компромиссы подробнее описаны далее в настоящей книге.

Другим очевидным способом сохранения времени программиста является "обучение машины" выполнять больше низкоуровневой работы по программированию, что приводит к формулировке следующего правила.

1.6.14. Правило генерации: избегайте кодирования вручную; если есть возможность, пишите программы для создания программ

Известно, что люди плохо справляются с деталями. Соответственно, любой вид ручного создания программ является источником задержек и ошибок. Чем проще и более абстрактной может быть программная спецификация, тем более вероятно, что проектировщик реализует ее правильно. Сгенерированный код (на

всех уровнях) почти всегда является более дешевым и более надежным, чем код, написанный вручную.

Общеизвестно, что это на самом деле так (в конце концов, именно поэтому созданы компиляторы и интерпретаторы), однако зачастую никто не задумывается о последствиях.

Изобилующий повторениями код на языке высокого уровня, написание которого утомительно для людей, является такой же продуктивной целью для генератора кода, как машинный код. Использование генераторов кода оправдано, когда они могут повысить уровень абстракции, т.е. когда язык спецификации для генератора проще, чем сгенерированный код, и код впоследствии не потребует ручной доработки.

В традициях Unix генераторы кода интенсивно используются для автоматизации чреватой ошибками кропотливой работы. Классическими примерами генераторов кода являются грамматические (parser) и лексические (lexer) анализаторы. Более новые примеры — генераторы make-файлов и построители GUI-интерфейсов.

Данные методики рассматриваются в главе 9.

1.6.15. Правило оптимизации: создайте опытные образцы, заставьте их работать, прежде чем перейти к оптимизации

Самый основной аргумент в пользу создания прототипов впервые был выдвинут Керниганом и Плоджером (Plauger): "90% актуальной и реальной функциональности лучше, чем 100% функциональности перспективной и сомнительной". Первоначальное создание прототипов помогает избежать слишком больших затрат времени для получения минимальной выгоды.

По несколько другим причинам Дональд Кнутт (автор книги

"Искусство компьютерного программирования" , одного из немногих действительно классических трудов в своей области) популяризовал мнение о том, что "преждевременная оптимизация — корень всех зол"[10], и он был прав.

Поспешная оптимизация до выяснения узких мест в программе, возможно, является единственной ошибкой, которая разрушила больше конструкций, чем излишнее наращивание функциональности. Искаженный код и непостижимое расположение данных — результат приоритета скорости и оптимизации использования памяти или дискового пространства за счет прозрачности и простоты. Они порождают бесчисленные ошибки и выливаются в миллионы человеко-часов — часто только для того, чтобы получить минимальную выгоду от использования какого-либо ресурса, гораздо менее дорогостоящего, чем время отладки.

Нередко преждевременная локальная оптимизация фактически препятствует глобальной оптимизации (и, следовательно, снижает общую производительность). Преждевременно оптимизированная часть конструкции часто смешивается с изменениями, которые имели бы гораздо больший выигрыш для конструкции в целом, поэтому проект завершается одновременно с низкой производительностью и чрезмерно сложным кодом.

В мире Unix имеется общепринятая и весьма явная традиция (ярко проиллюстрированная приведенными выше комментариями Роба Пайка и принципом Кена Томпсона о грубой силе), которая гласит:

создавайте прототипы, а затем шлифуйте их. Заставьте прототип работать, прежде чем оптимизировать его. Или:

сначала заставьте прототип работать, а потом заставьте его работать быстро. Гуру "Экстремального программирования" Кент Бек (Kent Beck), работающий в другой культуре, значительно усилил данный принцип: "заставьте программу работать, заставьте ее работать верно, а затем сделайте ее быстрой".

Сущность всех приведенных цитат одинакова: прежде чем пытаться регулировать программу, необходимо верно ее спроектировать и реализовать неоптимизированной, медленной и занимающей большой объем памяти. Затем ее следует регулировать систематически, определяя места, в которых можно получить большую производительность ценой наименьшего возможного повышения локальной сложности.

Создание прототипов так же важно для конструкции системы, как и оптимизация — гораздо проще определить, выполняет ли прототип возложенные на него задачи, чем читать длинную спецификацию. Я помню одного менеджера разработки в Bellcore, который боролся против культуры "требований" задолго до того, как все заговорили о "быстром создании прототипов" (fast prototyping), или "гибкой разработке" (agile development). Он не стал бы выпускать длинные спецификации. Он связывал сценарии оболочки и awk-код в некоторую комбинацию, которая делала приблизительно то, что ему было нужно, а затем просил заказчиков направить к нему нескольких клерков на пару дней. После чего приглашал заказчиков прийти и посмотреть, как их клерки используют прототип, и сказать ему, нравится им это или нет.

В случае положительного ответа он говорил, что "промышленный образец может быть создан через столько-то месяцев и по такой-то цене". Его оценки были довольно точными, однако он проигрывал в культуре менеджерам, которые верили, что создатели требований должны контролировать все. Майк Леск.

Использование прототипов для определения того, в каких функциях нет необходимости, способствует оптимизации производительности; нет необходимости оптимизировать то, что не написано. Наиболее мощным средством оптимизации можно назвать клавишу "Delete".

В один из наиболее продуктивных дней я удалил тысячу строк кода. Кен Томпсон.

Более глубоко соответствующие соображения рассматриваются в главе 12.

1.6.16. Правило разнообразия: не следует доверять утверждениям о "единственно верном пути"

Даже наилучшие программные инструменты ограничены воображением своих создателей. Никто не обладает умом, достаточным для оптимизации всего или для предвидения всех возможных вариантов использования создаваемой программы. Разработка негибкого, закрытого программного обеспечения, которое не будет сообщаться с остальным миром, является болезненной формой самолюбия.

Поэтому традиция Unix включает в себя здоровое недоверие к использованию "единственного верного пути" в проектировании или реализации программного обеспечения. В Unix допускается использование множества языков, открытых расширяемых систем и необходимой доработки.

1.6.17. Правило расширяемости: проектируйте с учетом изменений в будущем, поскольку будущее придет скорее, чем кажется

Если доверять заявлениям других людей о "единственном верном пути" неблагоразумно, еще большим безрассудством будет вера в непогрешимость собственных разработок. Никогда не считайте себя истиной в последней инстанции. Следовательно, оставляйте пространство для

роста форматов данных и кода. В противном случае часто будут возникать препятствия, связанные с прежними неразумными решениями, поскольку их невозможно изменить при поддержке обратной совместимости.

При проектировании протоколов или форматов файлов следует делать их достаточно самоописательными, для того чтобы их можно было расширять. Всегда,

всегда следует либо включать номер версии, либо составлять формат из самодостаточных, самоописательных команд так, чтобы можно было легко добавить новые директивы, а старые удалить, "не сбивая с толку" код чтения формата. Опыт Unix свидетельствует: минимальные дополнительные издержки, которые позволяют сделать расположение данных самоописательным, тысячекратно окупаются возможностью развивать его без разрушения конструкции.

Проектируя код, следует организовывать его так, чтобы будущие разработчики могли включать новые функции в архитектуру без необходимости ее перестройки, Данное правило не является разрешением на добавление функций, которые еще не нужны; это совет, призывающий писать код так, чтобы облегчить последующее добавление новых функций, когда они

действительно понадобятся. Необходимо сделать добавление новых функций гибким, а также вставлять в код комментарии, начинающиеся словами "если когда-нибудь потребуется сделать…". Это значительно облегчит работу тех, кто в последующем будет использовать и сопровождать код.

В их числе окажется и создатель программы, поддерживая код, который наполовину забудется под давлением более новых проектов. "Проектируя на будущее", можно сохранить собственное душевное равновесие.

1.7. Философия Unix в одном уроке

Вся философия в действительности сводится к одному железному правилу ведущих инженеров, священному "принципу KISS":

Unix предоставляет великолепную основу для применения принципа KISS. В последующих главах данной книги описано, как его следует применять.

1.8. Применение философии Unix

Описанные философские принципы не являются лишь неясными общими фразами. В мире Unix они приходят непосредственно из опыта и ведут к специфическим рецептам, часть из которых уже были сформулированы выше. Ниже приводится их список, не претендующий на полноту.

• Все, что может быть фильтром, независимым от отправителя и получателя,

должно быть таким фильтром.

• Потоки данных должны быть, если такое вообще возможно, текстовыми (для того чтобы их можно было просматривать и фильтровать с помощью стандартных средств).

• Структура баз данных и протоколы приложений должны быть, если это возможно, текстовыми (доступными человеку для чтения и редактирования).

• Сложные клиенты (пользовательские интерфейсы) должны быть четко отделены от сложных серверов.

• При возможности перед кодированием на C всегда нужно создавать прототип на каком-либо интерпретируемом языке.

• Смешивание языков лучше, чем написание всей программы на одном языке, тогда и только тогда, когда использование одного языка чрезмерно усложняет программу.

• Будьте "великодушны" к тому, что поступает, и строги к тому, что выпускается.

• При фильтровании не следует отбрасывать излишнюю информацию.

• Малое прекрасно. Следует писать программы, которые выполняют минимум, достаточный для решения задачи.

Далее в настоящей книге будут рассматриваться правила Unix-проектирования и следующие из них рецепты, применяемые снова и снова. Не удивительно, что они стремятся к объединению с наилучшими практическими приемами из программной инженерии в других традициях[11].

1.9. Подход также имеет значение

Когда верное решение очевидно, его следует немедленно использовать — данный подход может показаться работоспособным только в краткосрочной перспективе, однако это путь наименьших усилий в продолжительном периоде. Если не известно, какое решение является правильным, следует сделать минимум, необходимый для решения задачи, по крайней мере, до тех пор, пока не выяснится, что является правильным.

Для того чтобы правильно применять философию Unix, необходимо быть верным своему искусству. Необходимо верить, что проектирование программного обеспечения является искусством, достойным интеллектуальных и творческих усилий, терпения и настойчивости, на которые только способен разработчик. В противном случае разработчик не увидит в прошлом простых, стереотипных способов дизайна и реализации; он бросится кодировать, когда следовало бы подумать. Он будет небрежно усложнять, когда следовало бы неуклонно упрощать — а в дальнейшем удивится, почему код распухает, а отладка становится такой сложной.

Для того чтобы правильно применять философию Unix, необходимо ценить свое время и никогда не тратить его впустую. Если проблема была уже однажды решена кем-либо, не позволяйте амбициям или "политическим" соображениям втягивать себя в повторное ее решение вместо повторного использования. И никогда не работайте "тяжелее", чем требуется; вместо этого работайте изобретательнее и берегите дополнительные усилия на случай, когда они понадобятся. Опирайтесь на доступные средства и автоматизируйте все, что можно.

Проектирование и реализация программного обеспечения должно быть радостным

искусством, чем-то подобным интеллектуальной игре. Если эта позиция кажется читателю нелепой или весьма туманной, то ему следует остановиться, подумать и спросить себя, что упущено. Возможно, вместо разработки программ следует заняться чем-либо другим, чтобы зарабатывать деньги или убивать время. Программа должна стоить усилий ее разработчика.

Для того чтобы правильно применять философию Unix, необходимо прийти (или вернуться) к описанному подходу. Необходимо

интересоваться . Необходимо

экспериментировать . Необходимо усердно

исследовать .

Авторы надеются, что читатель при изучении оставшейся части книги воспользуется этим подходом, или, по крайней мере, что данная книга поможет открыть этот подход заново.

2

История: слияние двух культур

Тот, кто не помнит своего прошлого, обречен на его повторение.

Жизнь разума (The Life of Reason, 1905) —Джорж Сантаяна (George Santayana)

Прошлое освещает опыт. Операционная система Unix имеет долгую и колоритную историю, большая часть которой до сих пор живет в виде фольклора, предположений и (слишком часто) "боевых шрамов" в коллективной памяти Unix-программистов. Данная глава представляет собой обзор истории операционной системы Unix, который необходим для того, чтобы объяснить, почему в 2003 году современная Unix-культура выглядит именно так.

2.1. Истоки и история Unix, 1969–1995 гг.

Печально известный "эффект второй системы" (second-system effect) часто поражает преемников небольших экспериментальных прототипов. Стремление добавить все, что было отложено в ходе первой разработки, слишком часто ведет к большой и чрезмерно сложной конструкции. Менее широко известен, ввиду своего менее широкого распространения, эффект третьей системы. Иногда после того, как вторая система потерпела крах "под своим весом", существует возможность вернуться обратно к простоте и сделать все действительно правильно.

Первоначальная Unix была третьей системой. Ее прародителем была небольшая и простая система CTSS (Compatible Time-Sharing System — совместимая система разделения времени), она была либо первой, либо второй из используемых систем разделения времени (в зависимости от некоторых вопросов определения, которые вполне можно не учитывать). Систему CTSS сменил революционный проект Multics, попытка создать "информационную утилиту" с группами функций, которая бы изящно поддерживала интерактивное разделение

процессорного времени мэйнфреймов крупными сообществами пользователей. Увы, проект Multics рухнул "под собственным весом". Вместе с тем этот крах положил начало операционной системе Unix.

2.1.1. Начало: 1969–1971 гг.

Операционная система Unix возникла в 1969 году как детище разума Кена Томпсона, компьютерного ученого Bell Laboratories. Томпсон был исследователем в проекте Multics. Этот опыт утомлял Томпсона из-за примитивности пакетных вычислений, которые господствовали практически повсеместно. Однако в конце 60-х годов идея разделения времени все еще была новой. Первые предположения, связанные с ней, были выдвинуты почти десятью годами ранее компьютерным ученым Джоном Маккарти (John McCarthy) (который также известен как создатель языка Lisp), а первая фактическая реализация состоялась в 1962 году, семью годами ранее, и многопользовательские операционные системы все еще оставались экспериментальными и непредсказуемыми.

Аппаратное обеспечение компьютеров в то время было еще более примитивным. Наиболее мощные машины в те дни имели меньшую вычислительную мощность и внутреннюю память, чем обычный современный сотовый телефон[12]. Терминалы с видеодисплеями находились в начальной стадии своего развития и в течение последующих шести лет не получили широкого распространения. Стандартным интерактивным устройством на ранних системах разделения времени был телетайп ASR-33 — медленное, шумное устройство, которое печатало только символы верхнего регистра на больших рулонах желтой бумаги. ASR-33 послужит прародителем Unix-традиции в плане использования кратких команд и немногословных откликов.

Когда Bell Labs вышла из состава исследовательского консорциума Multics, у Кена Томпсона остались некоторые идеи создания файловой системы, вдохновленные проектом Multics. Кроме того, он остался без машины, на которой мог бы играть в написанную им игру "Space Travel" (Космическое путешествие), научно-фантастический симулятор управления ракетами в солнечной системе. Unix начала свой жизненный путь на восстановленном мини-компьютере PDP-7[13] в качестве платформы для игры Space Travel и испытательного стенда для идей Томпсона о разработке операционной системы. Подобный компьютер показан на рис. 2.1.

Рис. 2.1. Мини-компьютер PDP-7

Полная история происхождения описана в статье [69] с точки зрения первого помощника Томпсона, Денниса Ритчи, который впоследствии стал известен как соавтор Unix и создатель языка C. Деннис Ритчи, Дуг Макилрой и несколько их коллег привыкли к интерактивным вычислениям в системе Multics и не хотели терять эту возможность.

Ритчи отмечает: "То, что мы хотели сохранить, было не только хорошей средой для программирования, но и системой, вокруг которой могло сформироваться братство. Мы знали по опыту, что сущностью совместных вычислений, которые обеспечивались с помощью удаленного доступа к машинам с разделением времени, является не только ввод программ с клавиатуры в терминал вместо использования клавишного перфоратора, но и поддержка тесного общения". В воздухе витала идея рассматривать компьютеры не просто как логические устройства, а как ядра сообществ. 1969 год был также годом изобретения сети ARPANET (прямого предка современной сети Internet). Тема "братства" будет звучать во всей последующей истории Unix.

Реализация игры Томпсона и Ритчи "Space Travel" привлекла к себе внимание. Вначале программное обеспечение PDP-7 приходилось перекомпилировать на GE-мэйнфрейме. Программные утилиты, которые были написаны Томпсоном и Ритчи на самом PDP-7, чтобы поддерживать развитие игры, стали каркасом Unix, хотя само название не было закреплено за системой до 1970 года. Исходный вариант написания представлял собой аббревиатуру "UNICS" (UNiplexed Information and Computing Service). Позднее Ритчи описывал эту аббревиатуру как "слегка изменнический каламбур на слово "Multics", которое означало MULTiplexed Information and Computing Service.

Даже на самых ранних стадиях развития PDP-7 Unix была весьма похожа на современные Unix-системы и предоставляла гораздо более приятную среду программирования, чем доступные в то время мэйнфреймы с использованием перфокарт. Unix была весьма близкой к первой системе, используя которую программист мог находиться непосредственно перед машиной и создавать программы на лету, исследуя возможности и тестируя программы в процессе их создания. На протяжении всей жизни Unix реализуется модель развития возможностей путем привлечения высокоинтеллектуальных, добровольных усилий со стороны программистов, которых раздражают ограничения других операционных систем. Эта модель возникла на раннем этапе внутри самой корпорации Bell Labs.

Unix-традиция легковесной разработки и неформальных методов также началась вместе с появлением операционной системы. Тогда как Multics была крупным проектом с тысячами страниц технических спецификаций, написанных до появления аппаратного обеспечения, первый работающий код Unix был создан тремя коллегами путем мозгового штурма и реализован Кеном Томпсоном в течение двух дней на устаревшей машине, которая была задумана как графический терминал для "настоящего" компьютера.

Первой реальной задачей Unix в 1971 году была поддержка того, что в наши дни назвали бы текстовым процессором, для патентного департамента Bell Labs. Первым Unix-приложением был предшественник программы для форматирования текста

nroff(1). Этот проект оправдал приобретение гораздо более мощного мини-компьютера PDP-11. Руководство оставалось в счастливом неведении о том, что система обработки текста, которую создавали Томпсон и его коллеги, была инкубатором для операционной системы. Операционные системы не входили в планы Bell Labs — AT&T присоединилась к консорциуму Multics именно для того, чтобы избежать разработки собственной операционной системы. Тем не менее, завершенная система имела воодушевляющий успех, который утвердил Unix в качестве постоянной и ценной части системы вычислений в Bell Labs и привел к появлению другой темы в истории Unix — тесной связи со средствами набора и форматирования документов, а также средствами коммуникации. Справочник 1972 года говорил о 10 инсталляциях.

Позднее Дуг Макилрой напишет об этом периоде [53]: "Давление коллег и простая гордость своей квалификацией приводили к переписыванию или удалению больших фрагментов кода, по мере того, как появлялись лучшие или более фундаментальные идеи. Профессиональная конкуренция и защита сфер влияния были практически не известны: возникало так много хороших идей, что никому не требовалось быть собственником инноваций". Однако понадобится четверть века для того, чтобы сообществу стали ясны все выводы из этого наблюдения.

2.1.2. Исход: 1971–1980 гг.

Первоначально операционная система Unix была написана на ассемблере, а ее приложения

— на "смеси" ассемблера и интерпретируемого языка, который назывался "B". Его преимущество заключалось в том, что он был достаточно мал для работы на компьютерах PDP-7. Однако язык B не был достаточно мощным для системного программирования, поэтому Деннис Ритчи добавил в него типы данных и структуры, в результате чего появился язык C. Это произошло в начале 1971 года. А в 1973 году Томпсон и Ритчи наконец успешно переписали Unix на новом языке. Это был весьма смелый шаг. В то время для того чтобы получить максимальную производительность аппаратного обеспечения, системное программирование осуществлялось на ассемблере, и сама концепция переносимой операционной системы представлялась еще весьма сомнительной. Только в 1979 году Ритчи смог написать: "Кажется бесспорным, что в основном успех Unix обусловлен читаемостью, редактируемостью и переносимостью ее программного обеспечения, причем такие позитивные характеристики, в свою очередь, являются следствием ее написания на языках высокого уровня".

В 1974 году журнал

Communications of the ACM [72] впервые представил Unix широкой общественности. В статье авторы описывали беспрецедентно простую конструкцию Unix и сообщали о более чем 600 инсталляций данной операционной системы. Все инсталляции производились на машинах, мощность которых была низкой даже по стандартам того времени, однако (как писали Ритчи и Томпсон) "ограничения способствовали не только экономии, но также и определенному изяществу дизайна".

После доклада в CACM исследовательские лаборатории и университеты по всему миру заявили о желании испытать Unix самостоятельно. По соглашению сторон об урегулировании антитрастового дела 1958 года корпорации AT&T (родительской организации Bell Labs) запрещалось входить в компьютерный бизнес. Таким образом, Unix невозможно было превратить в продукт. Действительно, в соответствии с положениями соглашения, Bell Labs должна была лицензировать свою нетелефонную технологию всем желающим. Кен Томпсон без огласки начал отвечать на запросы, отправляя ленты и дисковые пакеты, каждый из которых, согласно легенде, подписывался "с любовью, Кен" (love, ken).

Напомним, что эти события происходили задолго до появления персональных компьютеров. Аппаратное обеспечение, необходимое для работы Unix, было слишком дорогостоящим, для того, чтобы использоваться в индивидуальных исследованиях. Поэтому Unix-машины были доступны, только благодаря благоволению больших организаций с большим бюджетом: корпораций, университетов, правительственных учреждений. Но использование таких мини-компьютеров было менее контролируемым, чем использование больших мэйнфреймов, и Unix-разработка быстро "приобрела дух контркультуры". Это было начало 70-х годов прошлого века. Первопроходцами Unix-программирования были лохматые хиппи и те, кто хотел на них походить. Они наслаждались, исследуя операционную систему, которая не только предлагала им интересные испытания на переднем крае компьютерной науки, но также и подрывала все технические предположения и бизнес-практику, которая сопутствовала "большим вычислениям". Перфокарты, язык COBOL, деловые костюмы и пакетные мэйнфреймы IBM остались в "презренном прошлом". Unix-хакеры упивались тем, что одновременно строили будущее и играли с системой.

Энтузиазм тех дней описывается цитатой Дугласа Комера (Douglas Comer): "Многие университеты вносили свой вклад в Unix. В Университете Торонто кафедра приобрела принтер/плоттер с разрешением 200 точек на дюйм и создала программное обеспечение, которое использовало его для имитации фотонаборного аппарата. В Йельском университете (Yale University) студенты и компьютерные ученые модифицировали командный интерпретатор Unix. В Университете Пурдью (Purdue University) кафедра электротехники добилась значительного улучшения производительности, выпустив версию Unix, которая поддерживала большее количество пользователей. В Пурдью также разработали одну из

первых компьютерных сетей на основе Unix. Студенты Калифорнийского университета в Беркли разработали новый командный интерпретатор и десятки небольших утилит. К концу 70-х годов, когда Bell Labs выпустила Version 7 UNIX, было ясно, что система решала вычислительные задачи многих факультетов и что она воплотила в себе множество появившихся в университетах идей. Конечным результатом стало появление укрепленной системы. Прилив идей послужил началом нового цикла интеллектуального взаимообмена академического мира и производственных лабораторий и в конечном итоге обусловил рост коммерческих вариантов Unix" [13].

Первой узнаваемой версией Unix была версия 7 (Version 7), выпущенная в 1979 году[14]. Первая группа пользователей Unix сформировалась за год до этого. К тому времени Unix использовалась для операционной поддержки всех систем Bell System [35], и получила распространение в университетах вплоть до Австралии, где заметки Джона Лайонза (John Lions) [49] в 1976 году по исходному коду Version 6 стали первой серьезной документацией по внутреннему устройству ядра Unix. Многие пожилые Unix-хакеры до сих пор свято хранят копии этих заметок.

Рис. 2.2. Кен Томпсон (сидя) и Деннис Ритчи перед PDP-11 в 1972 году

Книга Лайонза была самиздатовской сенсацией. Из-за несоблюдения авторских прав или чего-то еще ее нельзя было публиковать в США, поэтому везде "просачивались" копии копий. Я до сих пор храню свою, которая была копией как минимум шестого поколения. Не имея книги Лайонза, тогда невозможно было быть хакером ядра. Кен Арнольд.

Первые годы Unix-индустрии также были временем объединения. Первая Unix- компания (Santa Cruz Operation, SCO) начала свою работу в 1978 году, и в том же году продала первый коммерческий компилятор C (Whitesmiths). К 1980 году малоизвестная компания, производитель программного обеспечения в Сиэтле, также вступила в Unix-игру, распространяя вариант AT&T-версии Unix для мини- компьютеров, который назывался XENIX. Но привязанность Microsoft к Unix как к продукту не была долгой (хотя Unix использовалась для большей части внутренней разработки компании вплоть до начала 1990-х годов).

2.1.3. TCP/IP и Unix-войны: 1980–1990 гг.

Студенческий городок Калифорнийского университета в Беркли вначале был единственным важнейшим академическим центром Unix-разработки. Unix-исследо- вания начались здесь в 1974 году и получили мощный толчок, когда Кен Томпсон преподавал в университете в течение академического отпуска 1975-1976 годов. Первая BSD-версия была создана на базе лабораторной разработки в 1977 году неизвестным до этого аспирантом Биллом Джоем (Bill Joy). К 1980 году Беркли стал центром сети университетов, активно дополняющих свой вариант Unix. Идеи и код из Berkeley Unix (включая редактор

vi(1)) были переданы из Беркли обратно в Bell Labs.

Затем в 1980 году Агентству передовых исследований Министерства обороны США (Defense Advanced Research Projects Agency — DARPA) потребовалась команда для реализации новейшего набора протоколов TCP/IP на компьютерах VAX под Unix. Компьютеры PDP-10, поддерживающие сеть ARPANET, в то время устарели, и в воздухе уже витали идеи о том, что DEC может быть вынуждена отказаться от PDP-10 для поддержки VAX. Агентство DARPA рассматривало принятие DEC для реализации TCP/IP, однако отказалась от этой идеи, поскольку вызывал беспокойство тот факт, что DEC может не согласиться на изменения в

собственной операционной системе VAX/VMS [48]. Вместо этого агентство DARPA в качестве платформы выбрало Berkeley Unix, явно ввиду того, что ее исходный код был доступным и свободным [45].

Исследовательская группа компьютерных наук (Computer Science Research Group) в Беркли оказалась в нужное время в нужном месте с надежнейшими инструментами разработки. Это была, несомненно, наиболее важная поворотная точка в истории Unix с момента ее появления.

До того как реализация протокола TCP/IP была выпущена с версией BSD 4.2 в 1983 году, Unix имела только слабую поддержку сети. Ранние эксперименты с Ethernet были неудовлетворительными. Для распространения программного обеспечения через обычные телефонные линии посредством модема в Bell Labs было разработано неприглядное, но вполне функциональное средство, которое называлось UUCP (Unix to Unix Copy Program — программа копирования из Unix в Unix)[15]. С помощью UUCP можно было передавать Unix-почту между удаленными машинами. Кроме того, (после того, как в 1981 году была изобретена Usenet) UUCP поддерживала распределенные доски объявлений, которые позволяли пользователям широковещательно распространять текстовые сообщения везде, где были телефонные линии и Unix-системы.

Тем не менее, несколько Unix-пользователей, знакомых с яркими преимуществами ARPANET, испытывали существенные затруднения из-за отсутствия FTP и Telnet и очень ограниченного удаленного выполнения заданий и чрезвычайно медленных каналов. Ранее, до появления протоколов TCP/IP, Internet- и Unix-культуры не смешивались. Особое видение Деннисом Ритчи вопроса о компьютере как о способе "поддержки тесного общения" сформировало одно из университетских сообществ. Оно не расширилось до распределенной по всему континенту "сетевой нации", которую пользователи сети ARPA начали формировать в середине 70-х годов прошлого века. Первые пользователи ARPANET считали Unix грубой подделкой, "ковыляющей на смехотворно слабом аппаратном обеспечении".

После появления TCP/IP все изменилось. Началось слияние культур ARPANET и Unix. Это инициировало развитие, которое в конце концов спасло их от разрушения. Однако возникли новые проблемы, которые были результатом двух отдельных "бедствий": подъема Microsoft и падения AT&T.

В 1981 году корпорация Microsoft заключила свою историческую сделку с IBM по IBM PC. Билл Гейтс (Bill Gates) приобрел QDOS (Quick and Dirty Operating System), клон CP/M, собранный за шесть недель программистом Тимом Патерсоном (Tim Paterson), в Seattle Computer Products, где работал Тим Патерсон. Гейтс, скрывая от Патерсона и SCP сделку с IBM, приобрел права на систему за 50 тыс. долл. Затем он уговорил IBM разрешить Microsoft распространять на рынке операционную систему MS-DOS отдельно от аппаратного обеспечения PC. В течение следующего десятилетия выигрышный код, которого он не писал, сделал Билла Гейтса мультимиллиардером, а бизнес-тактика, еще более искусная, чем первоначальная сделка, принесли Microsoft монопольное положение на рынке настольных компьютеров. Операционная система XENIX как продукт была вскоре заброшена и в конце концов продана компании SCO.

В то время еще не было очевидно, насколько успешные (или деструктивные) действия собиралась предпринять Microsoft. Поскольку IBM PC-1 не обладала аппаратными возможностями поддержки Unix, пользователи Unix едва ли заметили эту платформу вообще (ирония в том, что операционная система DOS 2.0 затмила CP/M в основном из-за того, что один из основателей Microsoft Пол Аллен (Paul Allen) встроил в нее Unix-функции, включая подкаталоги и каналы). Рассмотрим события, которые кажутся более интересными, такие как возникновение в 1982 году компании Sun Microsystems.

Основатели Sun Microsystems, Билл Джой (Bill Joy), Андреас Бектолшейм (Andreas Bechtolsheim) и Винод Хосла (Vinod Khosla), начали создавать Unix-машину мечты со встроенной поддержкой сети. Они скомбинировали аппаратное обеспечение, спроектированное в Стэнфорде, с операционной системой Unix, разработанной в Беркли, и впоследствии основали индустрию рабочих станций. В то время никто не собирался контролировать доступ к исходному коду одной ветви дерева Unix, которая постепенно "высохла" в то время как Sun Microsystems начала вести себя все меньше как свободный проект и все больше как традиционная фирма. Университет в Беркли продолжал распространять BSD Unix вместе с исходным кодом. Официально лицензия на исходный код System III стоила 40 тыс. долл., однако Bell Labs закрывала глаза на рост количества распространяемых нелегальных лент с Bell Labs Unix, а университеты продолжали обмениваться кодом с Bell Labs. Складывалось впечатление, что коммерциализация Unix со стороны Sun — это лучшее, что могло произойти.

Также в 1982 году язык C впервые продемонстрировал признаки своего становления за пределами Unix-мира в качестве основного языка системного программирования. Всего через 5 лет C почти полностью вывел из употребления машинные ассемблеры. К началу 90-х годов языки C и C++ будут доминировать не только в системном, но и в прикладном программировании. К концу 90-х годов все остальные традиционные компилируемые языки фактически выйдут из употребления.

Когда в 1983 году DEC прекратила разработки по машине, которая являлась потомком PDP-10 (Jupiter), VAX-машины с использованием Unix стали занимать положение доминирующих Internet-машин, это положение они будут занимать до вытеснения их рабочими станциями Sun Microsystems. К 1985 году около 25% всех VAX-машин будут использовать Unix, причем несмотря на жесткое противостояние DEC. Однако самый долгосрочный эффект прекращения работ над проектом Jupiter был менее очевидным. Смерть хакерской культуры, развивавшейся вокруг компьютеров PDP-10 разработки MIT AI Lab, побудила программиста Ричарда Столлмена (Richard Stallman) к началу создания проекта GNU, полностью свободного клона Unix.

К 1983 году было не менее 6 Unix-подобных операционных систем для IBM-PC: uNETix, Venix, Coherent, QNX, Idris и вариант, поддерживаемый на низкоуровневой плате Sritek PC. Все еще не было вариантов ни версии System V, ни BSD-версии. Обе группы считали микропроцессор серии 8086 чрезвычайно маломощным и не планировали работать с ним. Ни одна из Unix-подобных операционных систем не добилась значительного коммерческого успеха, однако они показали значительную потребность в Unix на дешевом аппаратном обеспечении, которое ведущие производители оборудования не поставляли. Ни один человек не мог позволить себе приобрести лицензию на исходный код Unix.

Sun уже добилась успеха (с подражателями), когда в 1983 году Министерство юстиции США выиграло второй антитрастовый процесс против AT&T и раздробило корпорацию Bell System. Это освободило AT&T от соглашения 1958 года, препятствующего превращению Unix в продукт, и AT&T немедленно приступила к коммерциализации Unix System V. Этот шаг почти уничтожил Unix.

Это верно, но их маркетинг действительно распространил Unix по всему миру Кен Томпсон.

Большинство сторонников Unix полагали, что разделение AT&T было хорошей новостью. Нам казалось, что новые условия дадут шанс здоровой Unix-индустрии — индустрии, использующей недорогие рабочие станции на основе процессоров 68000, что в конечном итоге разрушит монополию IBM.

Никто из нас в то время не осознавал, что превращение Unix в коммерческий продукт разрушит свободный обмен исходным кодом, который дал столько жизненной силы

развивающейся системе. Не зная другой модели, кроме секретности для накопления прибыли от программного обеспечения и кроме централизованного управления разработкой коммерческого продукта, AT&T прекратила распространение исходного кода. Нелегальные ленты с Unix стали намного менее интересными, так как их использование могло повлечь за собой угрозу судебного преследования. Интеллектуальный вклад от университетов начал иссякать.

Осложняя положение, новые крупные игроки рынка Unix допускали большие стратегические ошибки. Одной из них было решение получить преимущество путем дифференцирования продуктов — тактика, которая в результате привела к тому, что в разных Unix-системах интерфейсы утратили идентичность. Это отбросило кроссплатформенную совместимость и фрагментировало рынок операционных систем Unix.

Другая менее очевидная ошибка заключалась в том, что все заинтересованные стороны вели себя так, будто персональные компьютеры и Microsoft не имеют отношения к перспективам Unix. В Sun Microsystems не разглядели того, что ставшие широко распространенными персональные компьютеры неизбежно начнут атаковать рынок рабочих станций Sun снизу. Корпорация AT&T, зациклившаяся на мини-компьютерах и мэйнфреймах, испытала несколько других стратегий, которые были направлены на то, чтобы стать главным игроком на рынке компьютеров, и крайне усугубила ситуацию. Для поддержки операционной системы Unix на PC-станциях были сформированы около десятка небольших компаний. Все они испытывали недостаток средств и уделяли основное внимание предоставлению своих услуг разработчикам и инженерам, т.е. никогда не ставили себе целью выход на рынок обслуживания предприятий и домашних пользователей, на который была нацелена Microsoft.

Действительно, в течение многих лет после падения AT&T Unix-сообщество было озабочено первой фазой Unix-войн — внутренними разногласиями и соперничеством между System V Unix и BSD Unix. Разногласия охватывали несколько уровней, ряд технических (сокеты против потоков, специальный терминальный интерфейс (tty) в BSD против общего терминального интерфейса (termio) в System V) и культурных вопросов. "Водораздел" пролегал приблизительно между хиппи и белыми воротничками. Программисты и технические специалисты склонялись к точке зрения Университета в Беркли и BSD Unix, а более ориентированные на бизнес специалисты — к точке зрения AT&T и System V Unix. Хиппи-программистам нравилось считать себя восставшими против корпоративной империи.

Однако в год разделения AT&T случилось нечто, что впоследствии приобрело более долгосрочную важность для Unix. Программист и лингвист по имени Ларри Уолл (Larry Wall) создал утилиту

patch(1) . Программа

patch — простой инструмент, который применяет к базовому файлу изменения, сгенерированные утилитой

diff(1) . Она предоставила Unix-разработчикам возможность сотрудничать, передавая друг другу вместо полных файлов наборы "заплат", т.е. инкрементальные изменения кода. Это было важно не только потому, что заплаты были менее громоздкими, чем полные файлы, но и потому, что они часто применяются аккуратно, даже если большая часть базового файла претерпела изменения с тех пор, как отправитель заплаты получил свою копию. С помощью данного средства потоки разработки по общему базовому исходному коду могли разделяться, двигаться параллельно и сходиться снова. Программа

patch сделала гораздо больше, чем любой другой инструмент, для активизации совместной разработки через Internet, т.е. активизации метода, который после 1990 года вдохнул в Unix новую жизнь.

В 1985 году корпорация Intel распространила первую микросхему 386-й серии, способную адресовать 4 Гбайт памяти с линейным адресным пространством. Неуклюжая сегментная адресация в процессорах 8086 и 286 очень быстро устарела. Это была важная новость, поскольку она означала, что впервые микропроцессор доминирующего семейства Intel стал способен работать под управлением Unix без болезненных компромиссов. Появление данного процессора предрекало грядущую катастрофу для Sun и других создателей рабочих станций, но они этого не заметили.

В том же году Ричард Столлмен опубликовал манифест GNU [78] и основал Фонд свободного программного обеспечения (Free Software Foundation). Очень немногие всерьез восприняли мнение Столлмена и его проект GNU, но он оказался прав. В независимой разработке того же года создатели системы X Window выпустили ее в виде исходного кода без указания права собственности, ограничений или лицензии. Как непосредственный результат этого решения возникла безопасно нейтральная зона для сотрудничества между Unix-поставщиками и потерпевшими поражение частными конкурентами с целью создания графической подсистемы Unix.

Кроме того, в 1983 году начались серьезные попытки по стандартизации согласования API-интерфейсов System V и Berkeley Unix вместе со стандартом /usr/group. За ним в 1985 году последовали стандарты POSIX, поддерживаемые институтом IEEE. В данных стандартах описывался пересекающийся набор вызовов BSD и SVR3 (System V Release 3), а также превосходная обработка сигналов Berkeley Unix и управление задачами, но с помощью терминального управления SVR3. Все последующие стандарты Unix будут базироваться на стандарте POSIX, а поздние Unix-системы будут строго его придерживаться. Единственным появившимся впоследствии главным дополнением к современному API-интерфейсу ядра Unix были BSD-сокеты.

В 1986 году Ларри Уолл, ранее разработавший утилиту

patch(1) , начал работу по созданию языка Perl, который станет первым и наиболее широко используемым языком написания сценариев с открытым исходным кодом. В начале 1987 года появилась первая версия GNU C-компилятора, а к концу того же года была определена основа инструментального набора GNU: редактор, компилятор, отладчик и другие базовые инструменты разработки. Тем временем система X Window начала появляться на относительно недорогих рабочих станциях. Эти компоненты в 90-х годах прошлого века послужили каркасом для Unix-разработок с открытым исходным кодом.

Также в 1986 году PC-технология освободилась от власти компании IBM, которая продолжала сохранять соотношение "цена-мощность" во всей линейке своих продуктов, что благоприятствовало ее высокодоходному бизнесу по поставке мэйнфреймов. IBM отказалась от процессора 386 в большей части своей новой линейки компьютеров PS/2 в пользу более слабого процессора 286-й серии. Серия PS/2, которую IBM разработала на основе частной архитектуры системной шины, для того чтобы оградить себя от создателей клонов, стала колоссально дорогим провалом[16]. Компания Compaq, самый активный создатель PC-клонов, превзошла IBM, выпустив первую машину с процессором 386. Даже при частоте процессора всего в 16 МГц, процессор 386-й серии сделал машину пригодной к использованию в среде Unix. Это был первый компьютер, который можно было назвать PC.

Проект Столлмена подтвердил возможность использовать машины 386-й серии для создания рабочих станций Unix по цене, почти на порядок меньшей, чем ранее. Как ни странно, видимо, никто в действительности не подумал об этом. Большинство Unix-программистов, "пришедших из мира мини-компьютеров и рабочих станций", продолжали пренебрегать дешевыми машинами 80x86, отдавая предпочтение более элегантным конструкциям на основе процессоров 68000. И хотя многие программисты способствовали успеху проекта GNU, большинство специалистов не видели его практических последствий в обозримом

будущем.

Unix-сообщество никогда не теряло своего бунтарского духа. Однако, анализируя прошлое, становится ясно, что приверженцы Unix также не смогли правильно оценить новые тенденции. Даже Ричард Столлмен, который за несколько лет до этого провозгласил моральный крестовый поход против частного программного обеспечения, действительно не понимал, насколько сильно превращение Unix в коммерческий продукт повредило сообществу, объединенному вокруг данной операционной системы. Идеи Столлмена были более абстрактными и касались долгосрочных проблем. Остальные члены сообщества продолжали надеяться, что некое рациональное изменение корпоративной формулы разрешит проблемы фрагментации, скверного маркетинга и стратегического направления, а также восстановит былые перспективы Unix. Но худшее было еще впереди.

В 1988 году Кен Олсен (Ken Olsen), президент корпорации DEC, "провозгласил" Unix чуть ли не панацеей. DEC поставляла собственный вариант Unix на компьютерах PDP-11 с 1982 года, однако в действительности рассчитывала перевести бизнес на частную операционную систему VMS. Корпорация DEC и индустрия мини-компьютеров столкнулись с крупными проблемами, их захлестнула волна мощных дешевых машин, продаваемых Sun Microsystems и остальными поставщиками рабочих станций. На большинстве этих рабочих станций использовалась Unix.

Однако собственные проблемы Unix-индустрии становились еще труднее. В 1988 году AT&T приобрела 20% акций Sun Microsystems. Две эти компании, лидеры Unix-рынка, начинали восставать против угрозы, созданной PC-станциями, корпорациями IBM и Microsoft, и осознавать, что предыдущие 5 лет "кровопролития" ослабили их. Альянс AT&T/Sun и разработка технических стандартов вокруг POSIX положили конец разногласиям между направлениями System V и BSD Unix. Однако началась "вторая фаза войн", когда поставщики второго уровня (IBM, DEC, Hewlett-Packard и другие) учредили Фонд открытого программного обеспечения (Open Software Foundation) и "выстроились против" линии AT&T/Sun (представленной международным сообществом Unix). Последовали новые этапы противостояния Unix против Unix.

Тем временем Microsoft зарабатывала миллиарды на рынке компьютеров для дома и малого бизнеса, на что воюющие стороны не пожелали обратить внимание. Выпуск в 1990 году Windows 3.0 — первой успешной операционной системы с графическим интерфейсом из Рэдмонда — закрепил доминирующее положение Microsoft и создал условия, которые позволили корпорации выровнять цены и монополизировать рынок настольных приложений 90-х годов.

Годы с 1989 по 1993 были самыми мрачными в истории Unix. Выяснилось, что мечты всего Unix-сообщества провалились. Междоусобная вражда чрезвычайно ослабила Unix-индустрию, почти лишила ее возможности противостоять Microsoft. Элегантные процессоры компании Motorola, которые предпочитали Unix-программисты, проиграли неказистым, но недорогим процессорам Intel. Проект GNU оказался неспособен выпустить свободное ядро Unix, что было обещано еще в 1985 году, и после нескольких лет отговорок доверие к нему стало падать. PC-технология была безжалостно превращена в коммерческий продукт. Пионеры хакерского движения 70-х годов достигли среднего возраста и сбавили темп. Аппаратное обеспечение дешевело, но Unix оставалась и дальше чрезмерно дорогой системой. Приверженцы Unix с опозданием поняли, что прежняя монополия IBM сменилась новой монополией Microsoft, а плохо сконструированное программное обеспечение Microsoft все больше заполняло рынок.

2.1.4. Бои против империи: 1991–1995 гг.

В 1990 году первую попытку противостояния сделал Вильям Джолитц (William Jolitz), опубликовав серию журнальных статей о перенесении BSD Unix на машины с процессором 386-й серии. Такой перенос был вполне возможным, так как, частично под влиянием Столлмена, хакер из университета в Беркли, Кит Бостик (Keith Bostic), в 1988 году начал удалять частный код AT&T из исходных файлов BSD. Однако проект 386BSD получил сильный удар, когда в конце 1991 года Джойлитц покинул проект и погубил собственную работу. Существует ряд противоречивых объяснений этому факту, однако все они имеют общую линию: Джойлитц хотел выпустить свой код как свободный, и был огорчен тем, что корпоративные спонсоры проекта выбрали более частную модель лицензирования.

В августе 1991 года Линус Торвальдс (Linus Torvalds), тогда еще неизвестный студент из Финляндии, объявил о проекте операционной системы Linux. Торвальдс вспоминает, что одним из главных мотивирующих факторов для него послужила высокая цена операционной системы Unix производства Sun Microsystems в его университете. Торвальдс также отмечал, что если бы он знал о проекте BSD Unix, то скорее присоединился бы к нему, чем создавал бы собственный проект. Однако проект 386BSD не распространялся до начала 1992 года, т.е. несколько месяцев спустя после появления первой версии Linux.

Важность обоих проектов стала очевидной только через несколько лет. В то время они мало привлекали внимание даже внутри культуры Internet-хакеров. Эти проекты оставались единичными внутри более широкого Unix-сообщества, которое все еще было зациклено на более производительных машинах, чем PC, и на попытках примирения особых свойств Unix с традиционной частной моделью бизнеса по созданию программного обеспечения.

Потребовалось еще 2 года и взрывной рост Internet в 1993-1994 годах, прежде чем истинная важность Linux и дистрибутивов BSD с открытым исходным кодом стала очевидной для остальной части мира Unix. К несчастью для приверженцев BSD, судебное преследование BSDI (начинающей компании, которая поддерживала проект Джойлитца) поглощало большую часть времени и побудило нескольких ключевых разработчиков Berkeley Unix переключиться на Linux.

В то время немало было сказано о копировании кода и краже фирменных секретов. Контрафактный код не был идентифицирован в течение приблизительно 2 лет. Судебный процесс мог бы продлиться еще дольше, если бы корпорация Novell не приобрела USL у AT&T и не урегулировала спор. В конце концов, из 18 000 файлов, составляющих дистрибутив, было удалено 3, а в другие файлы были внесены незначительные изменения. В дополнение к этому, университет согласился добавить указание на авторские права USL в почти 70 файлов при условии, что данные файлы и далее будут распространяться свободно. Маршал Кирк Маккьюзик.

Достигнутое соглашение создало важный прецедент, освободив полностью работающую Unix от частного контроля, однако его результаты для самой BSD Unix были крайне неприятными. Проблем не убавилось, когда в 1992-1994 годах Исследовательская группа компьютерных наук (Computer Science Research Group) в Беркли прекратила свою работу. Впоследствии междоусобное противостояние внутри BSD-сообщества разделилось на 3 конкурирующих проекта. В результате BSD-ветвь в решающее время осталась позади Linux и уступила данной операционной системе лидирующие позиции в Unix-сообществе.

Усилия разработчиков операционных систем Linux и BSD были естественными для Internet, чего нельзя сказать о предыдущих Unix-системах. Они опирались на распределенную разработку и инструмент Ларри Уолла (

patch(1)), а также привлекали разработчиков посредством электронной почты и групп

новостей Usenet. Соответственно, они получили огромный подъем, когда в 1993 году благодаря изменениям в телекоммуникационной технологии и приватизации магистральных Internet-каналов начали распространяться компании-провайдеры Internet-услуг. Этот процесс выходит за рамки описываемой истории. Потребность в дешевом Internet-доступе была вызвана другим фактором: изобретением в 1991 году технологии World Wide Web, которая была в Internet "приложением-приманкой" (killer app), технологией графического пользовательского интерфейса, которая сделала его дружественным для огромного числа нетехнических конечных пользователей.

Массовый маркетинг Internet одновременно увеличил число потенциальных разработчиков и сократил стоимость транзакций при распределенной разработке. Результаты проявились в проектах наподобие XFree86, в котором использовалась Internet-центрированная модель для создания более эффективной организации разработчиков, чем официальный Консорциум X (X Consortium). Первый выпуск сервера XFree86 в 1992 году обеспечил Linux и BSD подсистемой графического пользовательского интерфейса, которой им не доставало. В течение последующего десятилетия XFree86 будет лидировать среди X-разработок и основная деятельность Консорциума X будет состоять в отборе новаторских разработок, созданных в XFree86-сообществе, и направлении их обратно промышленным спонсорам консорциума.

К концу 1993 года операционная система Linux обладала как Internet-возможностями, так и системой X. Полный инструментарий GNU, обеспечивающий высококачественные средства разработки, поддерживался в Linux с самого начала. Более того, Linux была подобна чаше изобилия, притягивающей, накапливающей и концентрирующей 20 лет разработки программного обеспечения с открытым исходным кодом, которое до этого было рассеяно среди десятка различных частных Unix-платформ. Несмотря на то, что ядро Linux все еще официально представлялось бета-версией (уровня 0.99), оно было удивительно устойчивым. Размах и качество программного обеспечения в дистрибутивах Linux уже было таким же, как на действующих операционных системах.

Некоторые из Unix-разработчиков старой школы с более гибким мышлением стали отмечать, что долгожданная мечта о дешевой Unix-системе для каждого неожиданно начала воплощаться. Это происходило не благодаря AT&T, Sun или другому традиционному поставщику, и даже не благодаря организованным усилиям академических кругов. Это был бриколаж, созданный в Internet тем, что казалось спонтанным образованием, которое неожиданно заимствовало и комбинировало элементы Unix-традиции.

В стороне от этого процесса продолжалось корпоративное маневрирование. AT&T в 1992 году прекратила инвестировать Sun; затем в 1993 году продала свое подразделение Unix Systems Laboratories корпорации Novell. В 1994 году Novell передала торговую марку Unix группе разработки стандартов X/Open. В том же году AT&T и Novell присоединились к OSF, наконец положив конец Unix-войнам. В 1995 году компания SCO приобрела UnixWare (и права на оригинальные исходные коды Unix) у Novell. В 1996 году организации X/Open и OSF объединились, создав одну большую группу по разработке стандартов Unix.

Но традиционные поставщики Unix и последние сторонники противостояния казались все менее и менее значимыми. Активность и энергия в Unix-сообществе перемещалась к операционным системам Linux, BSD и разработчикам открытого исходного кода. К тому времени IBM, Intel и SCO анонсировали проект Monterey в 1998 году — последняя попытка объединить в одну большую систему все частные Unix-системы, оставшиеся "в стороне". Проект позабавил разработчиков и деловую прессу и внезапно прекратил существование в 2001 году после 3 лет движения в никуда.

Нельзя сказать, что промышленные транзакции были завершены до 2000 года, когда SCO продала UnixWare и оригинальные базовые исходные коды Unix компании Caldera,

производителю дистрибутивов Linux. Но после 1995 года история Unix стала историей движения открытого исходного кода. У этой истории есть и другой аспект, и для того чтобы описать его, необходимо вернуться к событиям 1961 года и возникновению культуры Internet-хакеров.

2.2. Истоки и история хакерской культуры, 1961–1995 гг.

Unix-традиция является скрытой культурой, а не просто набором технических приемов. Она передает совокупность ценностей, касающихся красоты и хорошего дизайна; в ней есть свои легенды и народные герои. С историей Unix-традиции переплелась другая неявная культура, четко обозначить которую еще труднее. В ней также есть собственные ценности, легенды и народные герои, частично совпадающие с традициями Unix, а частично унаследованные из других источников. Чаще всего ее называют "хакерской культурой", и с 1998 года она почти совершенно совпадает с тем, что в деловой компьютерной прессе называется "движением открытого исходного кода" (the open source movement).

Связи между Unix-традицией, хакерской культурой и движением открытого исходного кода неуловимые и сложные. Тот факт, что все три неявные культуры часто выражаются в поведении одних и тех же людей, не упрощает эти связи. Однако с 1990 года история Unix в значительной степени является историей того, как хакеры движения открытого исходного кода изменили правила и перехватили инициативу у частных поставщиков Unix старой линии. Таким образом, другая половина истории до современной Unix является историей хакеров.

2.2.1. Академические игры: 1961–1980 гг.

Корни хакерской культуры прослеживаются до 1961 года, когда в MIT (Massachusetts Institute of Technology — Массачусетский технологический институт) появился первый мини-компьютер PDP-1. PDP-1 был одним из ранних интерактивных компьютеров, и (в отличие от других машин) в то время был достаточно недорогим, поэтому время работы на нем жестко не регламентировалось. Он привлек к себе внимание группы любознательных студентов из клуба "Tech Model Railroad Club", которые из интереса проводили на нем эксперименты. В книге

"Hackers: Heroes of the Computer Revolution" [46] увлекательно описаны ранние дни клуба. Наиболее известным их достижением была "SPACEWAR" — игра, сюжет которой состоял в вольной трактовке космической оперы

"Lensman" Эдварда Эльмара "Док" Смита (E.E. "Doc" Smith)[17]

Несколько экспериментаторов из клуба TMRC позднее стали главными членами Лаборатории искусственного интеллекта Массачусетского технологического института (MIT Artificial Intelligence Lab), которая в 60-70-х годах прошлого века стала одним из мировых центров прогрессивной компьютерной науки. Они позаимствовали некоторые сленговые выражения и шутки клуба TMRC, в том числе традицию тонко (но безвредно) организовывать розыгрыши, которые назывались "hacks" ("шпильки"). Программисты лаборатории искусственного интеллекта, вероятнее всего, первыми стали называть себя "хакерами".

После 1969 года лаборатория MIT AI подключилась через сеть ARPANET к остальным

ведущим научно-исследовательским компьютерным лабораториям: в Стэнфорде, лаборатории компании Bolt Beranek & Newman, лаборатории Университета Карнеги-Меллон (Carnegie-Mellon University, CMU) и др. Исследователи и студенты впервые ощутили, как сеть с быстрым доступом стирала географические границы, способствовала сотрудничеству и дружбе.

Программное обеспечение, идеи, сленг и юмор передавались по экспериментальным каналам ARPANET. Начало формироваться нечто похожее на коллективную культуру. Так, многие помнят еще о т.н. "файле жаргона" (Jargon File) — списке широко используемых сленговых терминов, который возник в Стэнфорде в 1973 году и неоднократно перерабатывался в Массачусетском университете после 1976 года. Он аккумулировал в себе сленг из CMU, Йельского университета и других центров ARPANET.

Ранняя хакерская культура с технической точки зрения почти полностью поддерживалась мини-компьютерами PDP-10. На них использовались различные операционные системы, которые впоследствии вошли в историю: TOPS-10, TOPS-20, Multics, ITS, SAIL. Для программирования использовался ассемблер и диалекты LISP. Хакеры, работающие на PDP-10, взяли на себя поддержку самой сети ARPANET, поскольку других желающих выполнять эту работу не было. Позднее они стали основным кадровым составом IETF (Internet Engineering Task Force — инженерная группа по решению конкретной задачи в Internet) и основали традицию стандартизации посредством документов RFC (Requests For Comment — запросы на комментарии).

Это были исключительно талантливые молодые люди, чьим пристрастием стало программирование. Они склонялись к упорному отрицанию конформизма, спустя годы их будут называть "geeks" (помешанные). Они воспринимали компьютеры как устройства, создающие сообщество. Они читали Роберта Хейнлейна (Robert Heinlein) и Толкиена (J.R.R. Tolkien), играли в клубе "Общество творческого анахронизма" (Society for Creative Anachronism) и любили каламбуры. Несмотря на свои причуды (или, возможно, благодаря им), многие из них были в числе талантливейших программистов мира.

Они

не были Unix-программистами. Unix-сообщество произошло в основном из той же массы "помешанных" в академических кругах, правительственных или коммерческих исследовательских лабораториях, однако эти культуры имели важные различия, которые были обусловлены слабой поддержкой сети в ранней Unix. До начала 80-х годов доступ к сети ARPANET на основе Unix практически не осуществлялся, и индивидуумы, входящие одновременно в оба лагеря, встречались нечасто.

Совместная разработка и совместное использование исходного кода было ценной тактикой для Unix-программистов. Для ранних ARPANET-хакеров, с другой стороны, это было больше, чем тактика. Для них это было скорее чем-то подобным общей религии, частично возникшей из академической идеи "опубликуй или погибни" (publish or perish) и (в наиболее крайних версиях) развившейся в почти шарденистский идеализм сетевых интеллектуальных сообществ. Наиболее известным представителем этой среды хакеров стал Ричард М. Столлмен.

2.2.2. Internet и движение свободного программного обеспечения: 1981–1991 гг.

После возникновения BSD-варианта TCP/IP в 1983 году началось взаимопроникновение культур Unix и ARPANET. Это стало вполне логичным результатом появления

коммуникационных каналов, поскольку приверженцами обеих культур были в основном люди одного типа (а в действительности, зачастую

те же люди). ARPANET-хакеры изучали C и начали употреблять жаргонные слова: каналы, фильтры и оболочки. Unix-программисты осваивали TCP/IP и стали называть друг друга "хакерами". Процесс слияния ускорился после того, как закрытие проекта Jupiter в 1983 году уничтожило будущее компьютеров PDP-10. К 1987 году две культуры сблизились настолько, что большинство хакеров программировали на C и небрежно использовали жаргон клуба "Tech Model Railroad Club" 25-летней давности.

(Если в 1979 году еще казались необычными прочные связи в обеих культурах, в Unix и ARPANET, то уже к середине 80-х годов они никого не удивляли. Когда в 1991 году автор этой книги расширил старый файл жаргона ARPANET в

Новый словарь хакера (New Hacker's Dictionary) [66], две культуры практически смешались. Файл жаргона, созданный в ARPANET, но переработанный в Usenet, стал удачным символом этого слияния.)

Однако TCP/IP-сеть и сленг были не единственным наследием, которое после 1980 года хакерская культура получила благодаря своим "корням" в ARPANET. Эта культура стала неразрывно связана с именем Ричарда Столлмена.

Ричард M. Столлмен (широко известный под регистрационным именем RMS) к концу 1970 года уже доказал, что он является одним из наиболее способных современных программистов. Среди его многочисленных разработок был редактор Emacs. Для RMS закрытие проекта Jupiter в 1983 году только завершило дезинтеграцию культуры лаборатории MIT AI, которая началась несколькими годами ранее. RMS почувствовал себя изгнанным из хакерского рая и решил, что частное программное обеспечение достойно осуждения.

В 1983 году Столлмен основал проект GNU, целью которого было создание полностью свободной операционной системы. Хотя Столлмен никогда не был Unix-программистом, в условиях, создавшихся после 1980 года, реализация Unix-подобной операционной системы стала очевидной стратегией, которую следовало использовать. Большинство ранних помощников Столлмена были опытными ARPANET-хакерами, недавно влившимися в сообщество Unix.

В 1985 году Столлмен опубликовал Манифест GNU. В нем он сознательно создал идеологию из ценностей ARPANET-хакеров периода до 1980 года, полную новаторских этико-политических утверждений, независимых и характерных суждений. RMS ставил перед собой цель объединить рассеянное после 1980 года сообщество хакеров для достижения единой революционной цели. В его призывах явно прослеживались идеи Карла Маркса мобилизовать промышленный пролетариат против отчуждения результатов своего труда.

Манифест Столлмена разжег спор, который продолжается в хакерской среде и в наши дни. Его программа далеко выходила за рамки поддержки базового кода и, в сущности, подразумевала упразднение прав интеллектуальной собственности на программное обеспечение. Преследуя эту цель, Столлмен популяризировал понятие "свободное программное обеспечение" (free software), которое было первой попыткой определить продукт всей хакерской культуры. Столлмен написал Общедоступную лицензию (General Public License — GPL), которая должна была стать одновременно объединяющим началом и центром большой полемики по причинам, которые рассматриваются в главе 16. Дополнительные сведения, касающиеся позиции Столлмена и Фонда свободного программного обеспечения, приведены на Web-сайте проекта GNU <http://www.gnu.org>.

Понятие "свободное программное обеспечение" было частично описанием и частично попыткой определения культурной индивидуальности для хакеров. До Столлмена

представители хакерской культуры воспринимали друг друга как коллег-исследователей и использовали тот же сленг, однако никого не заботили споры о том, кем являются или должны быть "хакеры". После него в хакерской культуре стало ярче проявляться самосознание. Харизматическая и поляризованная фигура RMS оказалась в центре хакерской культуры, Столлмен стал ее героем и легендой. К 2000 году его уже едва ли можно было отличить от его легенды. В книге

"Free as in Freedom" [90] дана превосходная оценка его имиджа.

Аргументы Столлмена повлияли даже на поведение многих хакеров, которые продолжали скептически относиться к его теориям. В 1987 году он убедил попечителей BSD Unix в целесообразности удаления частного кода AT&T с целью выпустить свободную версию. Однако, несмотря на его решительные усилия, в течение более чем 15 лет после 1980 года хакерская культура так и не объединилась вокруг его идеологии.

Другие хакеры заново обнаруживали для себя открытые, совместные разработки без секретов по более прагматическим и менее идеологическим причинам. В конце 80-х годов, в нескольких зданиях офиса Ричарда Столлмена в Массачусетском технологическом университете, успешно трудилась группа разработки системы X. Она финансировалась поставщиками Unix, которые убедили друг друга быть выше проблем контроля и прав интеллектуальной собственности вокруг системы X Window, и не видели лучшей альтернативы, чем оставить ее свободной для всех. В 1987–1988-х годах разработка системы X послужила прообразом действительно крупных распределенных сообществ, которые пятью годами позже заново определят наиболее развитый участок исследований Unix.

Система X была одним из первых крупномасштабных проектов с открытым исходным кодом, разрабатываемым разнородным коллективом людей, работающих на различные организации и разбросанных по всему миру. К тому же электронная почта позволила быстро распространять идеи среди членов группы. Распространение программного обеспечения было вопросом нескольких часов, что позволяло согласованно функционировать всем участкам проекта. Сеть изменила способ разработки программного обеспечения. Кит Паккард.

X-разработчики не были последователями главного плана GNU, в то же время они не противодействовали ему активно. До 1995 года наиболее серьезное сопротивление плану GNU оказывали разработчики BSD. Приверженцы BSD, которые помнили, что они написали свободно распространяемое и модифицируемое программное обеспечение еще за годы до манифеста Столлмена, отвергали претензии проекта GNU на историческое и идеологическое главенство. Особенно они возражали против передающейся или "вирусной" собственности GPL, предлагая BSD-лицензию как "более свободную", поскольку она содержала меньше ограничений на повторное использование кода.

Это не помогало проекту Столлмена, и его попытки создания центральной части системы провалились, хотя его Фонд свободного программного обеспечения выпустил большую часть полного программного инструментария. Спустя 10 лет после учреждения проекта GNU, ядро GNU все еще не появилось. Несмотря на то, что отдельные средства, такие как Emacs и GCC, доказали свою потрясающую пользу, проект GNU без ядра не был способен ни угрожать гегемонии частных Unix-систем, ни предложить эффективное противодействие новой проблеме, связанной с монополией Microsoft.

После 1995 года спор вокруг идеологии Столлмена принял несколько иной оборот. Ее оппозиция стала ассоциироваться с именем Линуса Торвальдса, а также автора этой книги.

2.2.3. Linux и реакция прагматиков: 1991–1998 гг.

Даже когда проект HURD (GNU-ядро) "угасал", открывались новые возможности. В начале 90-х годов комбинация дешевых, мощных PC-компьютеров с простым Internet-доступом стала сильным соблазном для нового поколения молодых программистов, ищущих трудностей для испытания своего характера. Инструментарий пользователя, созданный Фондом свободного программного обеспечения, предложил весьма прогрессивный путь. Идеология следовала за экономикой, а не наоборот. Некоторые новички примкнули к крестовому походу Столлмена, приняв GNU в качестве его знамени, другим более импонировала традиция Unix в целом, и они присоединились к лагерю противников GPL, однако большинство вообще отказывалось от спора и просто писало код.

Линус Торвальдс искусно обошел противостояние, связанное с GPL, используя инструментальный набор GNU для окружения ядра Linux, созданного им, и лицензию GPL для его защиты, однако отказался от идеологической программы, которая следовала из лицензии Столлмена. Торвальдс заявлял, что в целом считает свободное программное обеспечение лучшим, чем частное, однако иногда использовал последнее. Его отказ становиться фанатиком даже в своем собственном деле делал его чрезвычайно привлекательным для большинства хакеров, которые чувствовали себя некомфортно под давлением разглагольствований Столлмена.

Жизнерадостный прагматизм Торвальдса и профессиональный, но скромный стиль, несомненно, способствовали удивительным победам хакерской культуры в 1993-1997 годах. Речь идет не только о технических успехах, но и о возникновении индустрии создания дистрибутивов, обслуживания и поддержки вокруг операционной системы Linux. В результате стремительно возрос его престиж и влияние. Торвальдс стал героем времени Internet. За 4 года (к 1995 году) он достиг таких "высот внутри культуры", для достижения которых Столлмену понадобилось 15 лет, причем он намного превзошел его рекорд в продаже "свободного программного обеспечения". В противоположность Торвальдсу, риторика Столлмена начала казаться резкой и неудачной.

Между 1991 и 1995 годами Linux перешла от тестовой среды, окружающей прототип ядра версии 0.1, к операционной системе, которая могла конкурировать по функциональности и производительности с частными Unix-системами, и превосходила их по таким важным характеристикам, как время непрерывной работы. В 1995 году Linux нашла свое приложение-приманку: Apache, Web-сервер с открытым исходным кодом. Как и Linux, Apache демонстрировал свою удивительную стабильность и эффективность. Linux-машины, поддерживающие Web-сервер Apache, быстро стали предпочтительной платформой для провайдеров Internet-услуг по всему миру; Apache поддерживает около 60% Web-сайтов[18], умело обыгрывая обоих своих главных частных конкурентов.

Единственное, чего не предложил Торвальдс, это новая идеология — новый рациональный или генеративный миф хакерского движения, позитивное суждение, заменяющее враждебность Столлмена к интеллектуальной собственности программой, более привлекательной для людей как внутри, так и вне хакерской культуры. Автор данной книги невольно восполнил этот недостаток в 1997 году, пытаясь понять, почему Linux-разработка несколько лет назад не была дезорганизована. Технические заключения опубликованных статей [67] приведены в главе 19. Здесь достаточно подчеркнуть, что при условии достаточно большого количества наблюдателей, все ошибки будут незначительными.

Это наблюдение подразумевает нечто такое, во что ни один представитель хакерской культуры не осмеливался действительно поверить в течение предыдущей четверти века, т.е. в надежные методы производства программного обеспечения, которое не просто более элегантно, но и более надежно, а значит,

лучше , чем код частных конкурентов. Этот вывод весьма неожиданно совпал с представлением Торвальдса о "свободном программном обеспечении". Для большинства хакеров и почти всех нехакеров девиз "создавайте свободные программы, поскольку они работают лучше" стал более привлекательным, чем девиз "создавайте свободные программы, потому что все программы должны быть свободными".

Контраст между "соборным" (т.е. централизованным, закрытым, контролируемым, замкнутым) и "базарным" (децентрализованным, открытым, с тщательной экспертной оценкой) видами разработки обусловил новое мышление данной среды. Это неразрывно связано с мнением Дуга Макилроя и Денниса Ритчи о братстве и обоюдном влиянии этого братства и ранних академических традиций ARPANET, связанных с экспертной оценкой и идеалистическими представлениями распределенного сообщества разума.

В начале 1998 года новое мышление подтолкнуло компанию Netscape Communications опубликовать исходный код браузера Mozilla. Внимание прессы к этому событию привело Linux на Уолл Стрит, способствовало возникновению бума акций технологических компаний в 1999-2000 годах, и действительно стало поворотной точкой, как в истории хакерской культуры, так и в истории Unix.

2.3. Движение открытого исходного кода: с 1998 года до настоящего времени

К моменту выхода браузера Mozilla в 1998 году хакерское сообщество наиболее правильно было бы охарактеризовать как множество группировок или братств. В него входили: движение свободного программного обеспечения Ричарда Столлмена, Linux-сообщество, Perl-сообщество, BSD-сообщество, Apache-сообщество, сообщество X-разработчиков, Инженерная группа по решению конкретной задачи в Internet (IETF) и как минимум десяток других объединений. Причем эти группировки пересекались, и отдельные разработчики, весьма вероятно, входили в состав двух или более групп.

Братство могло формироваться вокруг определенного базового кода, который поддерживался его членами, вокруг одного или нескольких харизматических лидеров, вокруг языка или средства разработки, вокруг определенной лицензии на программное обеспечение или технического стандарта, или вокруг организации- попечителя для некоторой части инфраструктуры. Наиболее почитаемой является группа IETF, которая неразрывно связана с появлением ARPANET в 1969 году. BSD-сообщество, традиции которого формировались в конце 70-х годов, пользуется большим престижем, несмотря на меньшее, чем у Linux, количество инсталляций. Движение свободного программного обеспечения Столлмена, возникновение которого датируется началом 80-х годов, относится к числу старших братств, как по историческому вкладу, так и в качестве куратора нескольких интенсивно и повседневно используемых программных инструментов.

После 1995 года Linux приобрела особую роль и как объединяющая платформа для большинства дополнительных программ сообщества, и как наиболее узнаваемое в среде хакеров имя. Linux-сообщество продемонстрировало соответствующую тенденцию поглощать другие братства, а, следовательно, выбирать и поглощать хакерские группировки, связанные с частными Unix-системами. Хакерская культура в целом начала объединяться вокруг общей миссии: как можно дальше продвинуть Linux и общественную (bazaar) модель разработки.

Поскольку хакерская культура после 1980 года "очень глубоко укоренилась" в Unix, новая миссия была неявным результатом триумфа Unix-традиции. Многие из старших лидеров хакерского сообщества, примкнувшие к Linux-движению, одновременно были ветеранами Unix, продолжающими традиции культурных войн 80-х годов после раздела AT&T.

Выход браузера Mozilla способствовал дальнейшему развитию идей. В марте 1998 года беспрецедентная встреча собрала влиятельных лидеров сообщества, представляющих почти все главные братства, для рассмотрения общих целей и тактики. В ходе этой встречи было принято новое определение общего для всех групп метода разработки: открытый исходный код (open source).

В течение 6 последующих месяцев почти все братства в хакерском сообществе приняли "открытый исходный код" как новое знамя. Более ранние группы, такие как IETF и BSD-разработчики, станут применять данное понятие ретроспективно к тому, что они делали до этого момента. Фактически к 2000 году риторика исходного кода будет не только объединять в хакерской культуре существующую практику и планы на будущее, но и представлять в новом свете свое прошлое.

Побудительный эффект от анонсирования продукта Netscape и новой важности Linux вышел далеко за пределы Unix-сообщества и хакерской культуры. Начиная с 1995 года, разработчики различных платформ, стоящих на пути Microsoft Windows (MacOS, Amiga, OS/2, DOS, CP/M, более слабые частные Unix-системы, различные операционные системы для мэйнфреймов, мини-компьютеров и устаревших микрокомпьютеров), сгруппировались вокруг языка Java, разработанного в Sun Microsystems. Многие недовольные Windows-разработчики присоединились к ним в надежде получить, по крайней мере, некоторую номинальную независимость от Microsoft. Однако поддержка Java со стороны Sun была (как описывается в главе 14) неумелой и разрозненной. Многим Java-разработчикам пришлось по вкусу то, что пропагандировалось в зарождающемся движении поддержки открытого исходного кода, и они вслед за Netscape перешли в сообщество Linux и открытого исходного кода, так же как ранее, следуя примеру Netscape, перешли к языку Java.

Активисты движения открытого исходного кода охотно приняли "волну иммигрантов" отовсюду. Старые приверженцы Unix стали разделять мечты новых иммигрантов о том, чтобы не просто пассивно мириться с монополией Microsoft, но и фактически осваивать ее рыночные секреты. Сообщество открытого исходного кода в целом подготовило основную поддержку господствующей тенденции и начало охотно вступать в альянс с главными корпорациями, в которых росли опасения потери контроля над собственным бизнесом ввиду более решительной и захватывающей тактики Microsoft.

А что же Ричард Столлмен и движение свободного программного обеспечения? Понятие "открытый исходный код" было явно предусмотрено для замены понятия "свободного программного обеспечения", которое предпочитал Столлмен. Он полусерьезно воспринимал данное понятие, а впоследствии отверг его на том основании, что оно не способно представить моральную позицию, которая была центральной с его точки зрения. С тех пор движение свободного программного обеспечения настаивает на своей обособленности от "открытого исходного кода", создавая, вероятно, наиболее значительный политический раскол в хакерской культуре последних лет.

Другой (и более важной) целью внедрения понятия "открытый исходный код" было представление методов хакерского сообщества остальному миру (в особенности лидирующим компаниям) более приемлемым и менее конфронтационным для рынка способом. В этой роли, к счастью, "открытый исходный код" добился безоговорочного успеха — привел к возрождению интереса к традициям Unix, из которых и произошло данное понятие.

2.4. Уроки истории Unix

Наиболее крупномасштабная модель в истории Unix представляет следующее: Unix процветала там и тогда, когда она наиболее близко приближалась к практике открытого исходного кода. Попытки сделать ее частной неизменно приводили к застою и упадку.

Анализируя прошлое, можно заключить, что это должно было стать очевидным гораздо раньше. Мы потеряли 10 лет после 1984 года, анализируя этот урок, и он, вероятно, является для нас слишком болезненным, чтобы его забыть.

То, что мы были более талантливыми, чем кто-либо другой в решении важных, но узких проблем разработки программного обеспечения, не спасло нас от почти полного непонимания взаимосвязи между технологией и экономикой. Даже самые перспективные и дальновидные мыслители Unix-сообщества были в лучшем случае полуслепыми. Урок на будущее состоит в том, что чрезмерная привязанность к какой- либо одной технологии или модели бизнеса была бы ошибочной, а поддержка адаптивной гибкости программного обеспечения и сопутствующей ему традиции проектирования является, соответственно, жизненно важной.

Другой урок: никогда не следует противостоять дешевому и гибкому решению. Иными словами, низкоклассная/массовая аппаратная технология почти всегда в конце концов поднимает кривую мощности и выигрывает. Экономист Клейтон Кристенсен (Clayton Christensen) называет это

пробивной технологией (disruptive technology) и в книге

'The Innovator's Dilemma" [12] демонстрирует, как это происходило с дисковыми накопителями, паровыми экскаваторами и мотоциклами. Мы видели это, когда мини-компьютеры заменили мэйнфреймы, рабочие станции и серверы сменили мини-компьютеры, а Intel-машины потребительского класса пришли на смену серверам и рабочим станциям. Движение открытого исходного кода выигрывает, потому что делает программное обеспечение потребительским товаром. Чтобы преуспеть, системе Unix необходимо выработать способность выбирать дешевое гибкое решение, а не пытаться бороться против него.

Наконец, Unix-сообщество старой школы потерпело неудачу в своих попытках быть "профессионалами", привлекая все управляющие механизмы традиционных корпоративных организаций, финансы и маркетинг.

3

Контраст: сравнение философии Unix и других операционных систем

Если ваша проблема выглядит неприступной, найдите пользователя Unix, который покажет, как ее решить.

Информационный бюллетень Дилберта (Dilbert), 3.0, 1994 —Скотт Адамс

Способы создания операционных систем, как очевидные, так и едва различимые, определяют стиль разработки программного обеспечения в них. Материал данной книги помогает лучше понимать взаимосвязи между конструкцией операционной системы Unix и развившейся вокруг нее философией проектирования программ. Поэтому считаем полезным представить сравнение стилей проектирования и программирования, свойственных системе Unix, и тех, которые характерны для остальных крупных операционных систем.

3.1. Составляющие стиля операционной системы

Прежде чем рассматривать те или иные операционные системы, необходимо определить "точку отсчета" для анализа способов, с помощью которых проектирование операционных систем может положительно или отрицательно повлиять на стиль программирования.

В целом, стили проектирования и программирования, связанные с различными операционными системами, вероятно, определяются тремя факторами: (a) замыслы разработчиков операционных систем, (b) единообразные формы, повлиявшие на конструкции посредством затрат и ограничений в среде программирования и (c) случайное культурное течение, ранние практические приемы, ставшие традиционными просто потому, что они были первыми.

Даже если принять как данность, что в каждом из сообществ, сформированных вокруг отдельных операционных систем, проявляются случайные культурные течения, оценка замыслов проектировщиков, а также затрат и ограничений, действительно позволяет обнаружить некоторые интересные модели. Сравнение этих моделей может способствовать более точному пониманию Unix-стиля. Выявить их можно путем анализа некоторых из наиболее важных различий операционных систем.

3.1.1. Унифицирующая идея операционной системы

В операционной системе Unix имеется несколько унифицирующих идей или метафор, которые формируют ее API-интерфейсы и определяемый ими стиль разработки. Наиболее важными из них, вероятно, являются модель, согласно которой "каждый объект является файлом", и метафора каналов (pipes)[19], построенная на ее поверхности. Как правило, стиль разработки в определенной операционной системе строго обусловлен унифицирующими идеями, которые были заложены в данную систему ее разработчиками. Они проникают в прикладное программирование из моделей, обеспеченных системными инструментами и API-интерфейсами.

Соответственно, при сравнении Unix с другой операционной системой, прежде всего, необходимо определить, имеет ли она унифицирующие идеи, которые формируют ее разработку, и если да, то чем они отличаются от идей Unix.

Для того чтобы разработать систему, абсолютно противоположную Unix, следует отказаться от унифицирующей идеи вообще, и иметь только несогласованную массу узкоспециализированных функций.

3.1.2. Поддержка многозадачности

Одним из основных отличий операционных систем является степень, с которой они способны поддерживать множество конкурирующих процессов. Операционная система самого низкого уровня (например DOS или CP/M), в сущности, представляет собой последовательный загрузчик программ без многозадачных возможностей. Операционные системы такого типа в

настоящее время не конкурентоспособны на неспециализированных компьютерах.

Для операционной системы более высокого уровня может быть характерна

невытесняющая многозадачность (cooperative multitasking) . Системы такого рода способны поддерживать множество процессов, однако при этом должно быть предусмотрено такое взаимодействие последних, когда один процесс произвольно "уступает" ресурсы процессора, прежде чем появится возможность запуска следующего процесса (таким образом, простые ошибки программирования могут привести к быстрой блокировке всей машины). Данный стиль операционных систем был временным переходным решением для аппаратного обеспечения, которое было достаточно мощным для поддержки конкурентных процессов, но испытывало либо недостаток периодических прерываний от таймера[20], либо недостаток в блоке управления памятью (memory-management unit), или то и другое. Многозадачность такого типа в настоящее время также является устаревшей и неконкурентоспособной.

В операционной системе Unix используется

вытесняющая многозадачность (preemptive multitasking) , при которой кванты времени распределяются системным планировщиком, который регулярно прерывает или вытесняет запущенный процесс для передачи управления следующему процессу. Почти все современные операционные системы поддерживают данный тип многозадачности.

Следует заметить, что понятие "многозадачная система" не тождественно понятию "многопользовательская система". Операционная система может быть многозадачной, но однопользовательской. В таком случае система используется для поддержки одной консоли и нескольких фоновых процессов. Истинная поддержка многопользовательской работы требует разграничений привилегий пользователей.

Для того чтобы создать систему, абсолютно противоположную Unix, вообще не следует поддерживать многозадачность или поддерживать, но испортить ее, окружая управление процессами множеством запретов, ограничений и частных случаев, при которых фактическое использование системы вне многозадачности весьма затрудняется.

3.1.3. Взаимодействующие процессы

В случае Unix малозатратное создание дочерних процессов (Process-Spawning) и простое межпроцессное взаимодействие (Inter-Process Communication — IPC) делают возможным использование целой системы небольших инструментов, каналов и фильтров. Данная система будет рассматриваться в главе 7; здесь необходимо указать некоторые последствия дорогостоящего создания дочерних процессов и IPC.

Канал был технически тривиален, но производил мощный эффект. Однако он никогда не стал бы простым без фундаментальной унифицирующей идеи процесса как автономной вычислительной единицы с программируемым управлением. В операционной системе Multics командный интерпретатор представлял собой просто другой процесс; управление процессами "не пришло свыше" вписанным в JCL. Дуг Макилрой.

Если операционная система характеризуется дорогостоящим созданием новых процессов и/или управление процессами является сложным и негибким, то, как правило, проявляются описанные ниже последствия.

• Более естественным способом программирования становятся монолитные гигантские

конструкции.

• Большая часть политики должна быть реализована внутри этих монолитов. Это подталкивает к использованию C++ и замысловатой многослойной внутренней организации кода, вместо C и относительно простых внутренних иерархий.

• Когда процессы не могут избежать взаимодействия, они производят обмен данными посредством механизмов, которые являются громоздкими, неэффективными и небезопасными (например, с помощью временных файлов), или путем сохранения чрезмерно большого количества сведений о реализациях других процессов.

• Мультипроцессная обработка (multithreading) широко применяется для задач, которые в Unix обрабатывались бы с помощью множества сообщающихся друг с другом легковесных процессов.

• Возникает необходимость изучения и использования асинхронного ввода-вывода.

Существует ряд примеров общих стилистических особенностей (даже в прикладном программировании), которые возникают в связи с ограничениями операционной системы.

Неочевидным, но важным свойством каналов и других классических IPC-методов Unix является то, что они требуют такой уровень простоты обмена данными между программами, который побуждает разделение функции. Напротив, отсутствие эквивалента каналов проявляется в том, что взаимодействие программ может быть реализовано только путем внедрения в них полного объема сведений о внутреннем устройстве друг друга.

В операционных системах без гибкого IPC-механизма и прочной традиции его использования программы обмениваются данными путем совместного применения сложных структур данных. Поскольку проблему связи необходимо решать заново для всех программ каждый раз при добавлении новой программы в набор, сложность данного решения возрастает как квадрат числа взаимодействующих программ. Еще хуже то, что любое изменение в одной из открытых структур данных может вызвать неочевидные ошибки в неопределенно большом числе других программ.

Word, Excel, PowerPoint и другие программы Microsoft обладают детальными, можно сказать безграничными, знаниями о внутреннем устройстве друг друга. В Unix программист пытается разрабатывать программы не только для взаимодействия друг с другом, но и с еще не созданными программами. Дуг Макилрой.

Данная тема также затрагивается в главе 7.

Для создания системы, абсолютно противоположной Unix, следует делать создание дочерних процессов очень дорогостоящим, управление процессами сложным и негибким, и оставлять IPC-механизм как неподдерживаемый или как наполовину поддерживаемое запоздалое решение.

3.1.4. Внутренние границы

В Unix действует предположение о том, что программист знает лучше (чем система). Система не остановит пользователя и не потребует какого-либо подтверждения при выполнении опасных действий с данными, таких как ввод команды rm -rf *. С другой стороны, Unix весьма заботится о том, чтобы не допустить одного пользователя к данным другого. Фактически Unix побуждает пользователя иметь несколько учетных записей, в каждой из которых имеются

собственные и, возможно, отличные от других записей привилегии, позволяющие пользователю обезопасить себя от аномально работающих программ[21]. Системные программы часто обладают собственными учетными записями псевдопользователей, которые нужны для присвоения прав доступа только к определенным системным файлам без необходимости неограниченного доступа (или доступа с правами

суперпользователя (superuser)).

В Unix имеется по крайней мере три уровня внутренних границ, которые защищают от злонамеренных пользователей или чреватых ошибками программ. Одним из таких уровней является управление памятью. Unix использует аппаратный блок управления памятью (Memory Management Unit — MMU), который препятствует вторжению одних процессов в адресное пространство памяти других. Вторым уровнем является реальное присутствие групп привилегий для множества пользователей. Процессы обычных пользователей (т.е. не администраторов) не могут без разрешения изменять или считывать файлы других пользователей. Третьим уровнем является заключение критичных к безопасности функций в минимально возможные участки благонадежного кода. В Unix даже оболочка (системный командный интерпретатор) не является привилегированной программой.

Целостность внутренних границ операционной системы является не просто абстрактным вопросом дизайна. Она имеет важные практические последствия для безопасности системы.

Чтобы разработать систему, полностью противоположную Unix, необходимо отказаться от управления памятью, с тем чтобы вышедший из-под контроля процесс мог разрушить любую запущенную программу или же заблокировать и повредить ее. В этом случае следует поддерживать слабые группы привилегий или не иметь их вообще, с тем чтобы пользователи могли без труда изменять чужие файлы и важные системные данные (например, макровирус, захвативший контроль над текстовым процессором, может отформатировать жесткий диск). Кроме того, следует доверять большим блокам кода, подобным оболочке и GUI-интерфейсу, так чтобы любая ошибка или успешная атака на данный код становилась угрозой для всей системы.

3.1.5. Атрибуты файлов и структуры записи

Unix-файлы не имеют ни структур записи (record structure), ни атрибутов. В некоторых операционных системах файлы имеют связанные структуры записи; операционная система (или ее служебные библиотеки) "знает" о файлах с фиксированной длиной записи или об ограничивающем символе текстовой строки, а также о том, следует ли читать последовательность CR/LF как один логический символ.

В других операционных системах файлы и каталоги имеют связанные с ними пары имя/атрибут — внешние данные, используемые (например) для связи файла документа с распознающим его приложением. (Классический способ поддержки таких связей в Unix заключается в том, чтобы заставить приложения опознавать "магические числа" или другие типы данных, находящиеся внутри самого файла.)

Одна из проблем оптимизации связана со структурами записи уровня операционной системы. Это гораздо более серьезная проблема, чем простое усложнение API-интерфейсов и работы программистов. Они подталкивают программистов использовать неясные форматы файлов, ориентированных на записи, которые невозможно соответствующим образом прочитать с помощью обычных инструментов, таких как текстовые редакторы.

Атрибуты файлов могут быть полезными, однако (как будет сказано в главе 20) они способны создавать некоторые каверзные проблемы семантики в среде каналов и инструментов, ориентированных на байтовые потоки. Когда атрибуты файлов поддерживаются на уровне операционной системы, они побуждают программистов использовать неясные форматы и опираться на атрибуты файлов для их связывания со специфическими интерпретирующими приложениями.

Для разработки системы, совершенно противоположной Unix, необходимо иметь громоздкий набор структур записи, которые заставят угадывать, способен ли какой- либо определенный инструмент прочесть файл в таком виде, каким он был создан в исходном приложении. Добавляйте файловые атрибуты и создавайте систему, сильно зависимую от них, с тем чтобы семантику файла нельзя было определить путем изучения данных внутри данного файла.

3.1.6. Двоичные форматы файлов

Если в операционной системе применяются двоичные форматы для важных данных (таких как учетные записи пользователей), вполне вероятно, что традиции использования читабельных текстовых форматов для приложений не сформируются. Более подробно о том, почему данный подход является проблемным, будет сказано в главе 5. Здесь достаточно упомянуть о нескольких последствиях.

• Даже если поддерживается интерфейс командной строки, написание сценариев и каналы, в системе будут развиваться только очень немногие фильтры.

• Доступ к файлам данных можно будет получить только посредством специальных средств. Для разработчиков главными станут данные средства, а не файлы данных. Следовательно, различные версии форматов файлов часто будут несовместимыми.

Для проектирования системы, полностью противоположной Unix, нужно сделать все форматы файлов двоичными и неясными, а также сделать обязательным использование тяжеловесных инструментов для их чтения и редактирования.

3.1.7. Предпочтительный стиль пользовательского интерфейса

В главе 11 подробно рассматриваются различия между

интерфейсами командной строки (Command-Line Interfaces — CLI) и

графическими пользовательскими интерфейсами (Graphical User Interfaces — GUI) . Выбор проектировщиком операционной системы одного из этих типов в качестве обычного режима представления влияет на многие аспекты конструкции от планирования процессов и управления памятью до

программных интерфейсов приложений (Application Programming Interfaces — API) , предоставленных приложениям для использования.

С момента появления первых компьютеров Macintosh понадобилось достаточно много лет, чтобы специалисты убедились в том, что слабые средства GUI-интерфейса в операционной системе являются проблемой. Опыт Unix противоположен: слабые средства CLI-интерфейса

представляют собой менее очевидный, но не менее серьезный недостаток.

Если в операционной системе CLI-средства являются слабыми или их вообще нет, то проявляются описанные ниже последствия.

• Никто не будет разрабатывать программы, взаимодействующие друг с другом неожиданным способом, поскольку это будет

невозможно. Вывод одной программы невозможно будет использовать в качестве ввода другой.

• Удаленное системное администрирование будет слабым и трудным в использовании, и будет потреблять больше сетевых ресурсов[22].

• Даже простые неинтерактивные программы будут нести на себе издержки графического интерфейса или замысловатого интерфейса сценариев.

• Программирование любым изящным способом серверов, системных служб и фоновых процессов будет, вероятно, невозможным или, по крайней мере, значительно затруднится.

Для разработки системы, полностью противоположной Unix, нужно отказаться от CLI-интерфейса и возможностей включения программ в сценарии, или использовать важные средства, управлять которыми с помощью CLI-интерфейса невозможно.

3.1.8. Предполагаемый потребитель

Дизайн той или иной операционной системы прямо зависит от ее потребителя. Некоторые операционные системы предназначены для лабораторий, другие — для настольных компьютеров. Одни системы разрабатываются для технических специалистов, иные — для конечных пользователей. Некоторые предназначены для обособленной работы в управляющих приложениях реального времени, другие — для работы в качестве среды для разделения времени и распределенных сетей.

К очень важным отличительным факторам относятся клиент и сервер. "Клиент" представляет собой легковесную систему, поддерживающую только одного пользователя. Такая система способна работать на небольших машинах и предназначена для включения в случае необходимости и отключения после завершения пользователем работы. Она не имеет вытесняющей многозадачности, оптимизирована по низкой задержке и выделяет большую часть своих ресурсов для поддержки вычурных пользовательских интерфейсов. "Сервер" — тяжеловесная система, способная работать продолжительное время и оптимизированная по пропускной способности. Для поддержки множества сеансов в такой системе используется полная вытесняющая многозадачность. Первоначально все операционные системы были серверными. Идея клиентской операционной системы возникла только в конце 70-х годов прошлого века с появлением PC, недорогого аппаратного обеспечения невысокой мощности. В клиентских операционных системах основное внимание уделяется визуально привлекательному для пользователя внешнему виду, а не бесперебойной работе 24 часа в сутки 7 дней в неделю.

Кроме того, на стиль разработки, конечно же, влияет допустимый с точки зрения целевой аудитории уровень сложности интерфейса и то, как эта сложность соотносится со стоимостью и производительностью. Об операционной системе Unix часто говорят, что она создана программистами для программистов, т.е. для целевой аудитории, которая известна своей

терпимостью к сложности интерфейса.

Это скорее следствие, чем цель. Я испытываю отвращение к системе, разработанной для "пользователя", если в слове "пользователь" закодировано уничижительное значение "тупой и примитивный". Кен Томпсон.

Для того чтобы разработать операционную систему, абсолютно противоположную Unix, ее нужно писать так, как будто она "знает" о намерениях пользователя больше, чем он сам.

3.1.9. Входные барьеры для разработчика

Другой важной характеристикой, по которой различают операционные системы, является совокупность сложностей, препятствующих простым пользователям стать разработчиками. Существует два определяющих фактора. Одним из них является денежная стоимость средств разработки, а другим — затраты времени, необходимые для того, чтобы развить мастерство разработчика. В некоторых средах развиваются также социальные барьеры, однако они обычно являются следствием базовых технологических сложностей, а не первопричиной.

Дорогостоящие инструменты разработки и сложные неясные API-интерфейсы ведут к возникновению небольших элитных культур программирования. В таких культурах программные проекты являются крупными, каковыми они и должны быть, для того чтобы окупить вложения как финансового, так и интеллектуального (человеческого) капитала. Для крупных проектов характерно создание крупных программ (и, как следствие, это часто приводит к большим дорогостоящим провалам).

Недорогие инструменты и простые интерфейсы поддерживают любительское программирование, культуру увлеченных энтузиастов и исследования. Программные проекты могут быть небольшими (часто формальная структура проекта является явно излишней), а провалы не являются катастрофическими. Это меняет стиль разработки кода. Кроме прочих преимуществ, они демонстрируют меньшую склонность к неверным подходам.

Любительское программирование стремится к созданию большого количества небольших программ и сообщества знаний, которое самостоятельно укрепляется и расширяется. В мире дешевого аппаратного обеспечения присутствие или отсутствие такого сообщества становится все более важным фактором, определяющим, будет ли операционная система жизнеспособной в течение длительного времени.

Любительское программирование зародилось в Unix. Одним из новшеств, которое впервые появилось в Unix, была поставка компилятора и инструментов написания сценариев как части стандартного инсталляционного набора, доступного для всех пользователей. Это поддерживало культуру разработки программного обеспечения как хобби, которая охватила множество инсталляций. Множество любителей, писавших код в Unix, не считали свое занятие разработкой кода, они считали его написанием сценариев для автоматизации общих задач или настройкой своей среды.

Для того чтобы разработать систему, полностью противоположную Unix, нужно сделать любительское программирование невозможным.

3.2. Сравнение операционных систем

Логика выбора конструкции Unix становится более очевидной в сравнении с другими операционными системами. Ниже приводится только общий обзор конструкций[23].

На рис. 3.1. отражены генетические связи между рассматриваемыми операционными системами разделения времени. Несколько других операционных систем (отмеченные серым цветом и не обязательно являющиеся системами разделения времени) включены для расширения контекста. Системы, названия которых обрамлены сплошными линиями, до сих пор существуют. Дата "рождения" представляет собой дату первой поставки[24], дата "смерти", как правило, — это дата, когда поставщик прекратил поставку системы.

Сплошные стрелки указывают на генетическую связь или очень сильное влияние дизайна (т.е. более позднюю систему с API-интерфейсом, умышленно переработанным путем обратного проектирования для соответствия более ранней системе). Штриховые линии указывают на значительное влияние конструкции, а пунктирные — наоборот, на слабое влияние конструкции. Не все генетические связи подтверждаются разработчиками. В действительности, некоторые из них были официально отвергнуты по соображениям законности или корпоративной стратегии, но являются открытыми секретами в индустрии.

Блок "Unix" включает в себя все частные Unix-системы, включая AT&T и ранние версии Berkeley Unix. Блок "Linux" включает в себя Unix-системы с открытыми исходными кодами, каждая из которых была основана в 1991 году. Они имеют генетическую наследственность от раннего Unix-кода, который был освобожден от частного контроля AT&T соглашением по судебному процессу 1993 года[25].

3.2.1. VMS

VMS — частная операционная система, первоначально разработанная для мини-компьютера VAX корпорации "Digital Equipment Corporation" (DEC). Впервые она была выпущена в 1978 году и была важной действующей операционной системой в 80-х и начале 90-х годов. Сопровождение данной системы продолжалось, когда DEC была приобретена компанией Compaq, а последняя — корпорацией Hewlett-Packard.

На момент написания книги операционная система VMS продолжала продаваться и поддерживаться[26]. VMS приведена в данном обзоре для демонстрации контраста между Unix и другими CLI-ориентированными операционными системами эры мини-компьютеров.

Рис. 3.1. Схема исторических связей между системами разделения времени

Операционная система VMS имеет полную вытесняющую многозадачность, однако создание дочерних процессов в ней весьма дорогостоящее. Файловая система в VMS имеет детально разработанное понятие типов записи (хотя в ней нет атрибутов). Данные черты приводят к тем же последствиям, которые рассматривались выше, в особенности в VMS проявляется тенденция к увеличению размеров программ и созданию тяжеловесных монолитов.

VMS характеризуется длинными, четкими системными командами, подобными инструкциям COBOL, и параметрами команд. В VMS имеется весьма полная интерактивная справочная система (не по API-интерфейсам, а по запускаемым программам и синтаксису командной строки). Фактически CLI-интерфейс VMS и ее справочная система являются организационной метафорой VMS. Хотя система X Window модифицирована для VMS, подробный CLI-интерфейс продолжает оказывать наиболее важное стилистическое влияние на

проектирование программ. В связи с этим определяется ряд следующих факторов и последствий.

• Частота, с которой используются функции командной строки — чем длиннее команда, которую необходимо ввести, тем меньше пользователь хочет это делать.

• Размер программ — люди хотят вводить с клавиатуры меньше команд, а значит, использовать меньше программ и писать более крупные программы с большим количеством функций.

• Количество и тип принимаемых программой параметров — они должны соответствовать синтаксическим ограничениям, налагаемым справочной системой.

• Простота использования справочной системы — справка в VMS весьма полная, но поиск и поисковые средства в ней отсутствуют, кроме того, индексация справочной системы недостаточная. Это затрудняет получение четких сведений, поддерживает специализацию и препятствует любительскому программированию.

VMS имеет заслуживающую доверия систему внутренних границ. Она была разработана для действительно многопользовательской работы, и для того чтобы оградить процессы друг от друга, полностью использует аппаратный блок MMU. Системный интерпретатор команд является привилегированной программой, но инкапсуляция важных функций остается достаточно хорошей. Взломы системы безопасности VMS бывают редко.

Первоначально VMS-инструменты были дорогими, а интерфейсы сложными. Огромное количество программной документации для VMS доступны только в бумажной форме, поэтому поиск каких-либо сведений является продолжительной и трудоемкой операцией. Данные причины препятствовали исследовательскому программированию и изучению обширного инструментария. Только после того как поставщик VMS почти забросил данную систему, вокруг нее развилось любительское программирование и культура хобби, но данная культура не является особенно стойкой.

Подобно Unix, операционная система VMS предшествовала разграничению клиент/сервер. Она была успешной в свое время в качестве общецелевой операционной системы разделения времени. Целевую аудиторию главным образом представляли технические пользователи и преимущественно программные предприятия, допускающие умеренную сложность.

3.2.2. MacOS

Операционная система Macintosh была разработана в компании Apple в начале 80-х годов прошлого века. Ее создателей вдохновила передовая работа по разработке GUI-интерфейсов, осуществленная ранее в Исследовательском центре Palo Alto (Palo Alto Research Center) компании Xerox. Она увидела свет вместе с Macintosh в 1984 году. С тех пор MacOS подверглась двум значительным преобразованиям конструкции, а в настоящее время претерпевает третье. Первое преобразование было связано с переходом от поддержки только одного приложения в тот или иной момент времени к невытесняющей многозадачности (MultiFinder). Вторым преобразованием был переход с процессоров серии 68000 на процессоры PowerPC, что позволило сохранить обратную бинарную совместимость с приложениями 68K, а также добавило для PowerPC-приложений усовершенствованную систему управления общими библиотеками, заменяющую исходную систему прерываний совместно используемых программ на основе инструкций процессора 68K. Третьим

преобразованием было объединение в системе MacOS X конструкторских идей MacOS с Unix-производной инфраструктурой. В данном разделе рассматриваются предшествующие MacOS X версии данной системы, кроме случаев, где это будет отмечено особо.

В MacOS прослеживается очень сильное влияние унифицирующей идеи, которая весьма отличается от идеи Unix: нормы проектирования интерфейсов компьютеров Macintosh (Mac Interface Guidelines). Они подробнейшим образом определяют внешний вид графического интерфейса приложений и режимы его работы. Согласованность норм значительно влияет на культуру пользователей компьютеров Macintosh. Нередко просто перенесенные программы из DOS или Unix, не соблюдающие определенные нормы, немедленно отвергаются сообществом Mac-пользователей и терпят неудачи на рынке.

Одна из ключевых идей относительно норм заключается в том, что рабочие компоненты должны находиться там, куда их перенес пользователь. Документы, каталоги и другие объекты постоянно сохраняются на рабочем столе, не смешиваясь с системной информацией, а содержание рабочего стола сохраняется при перезагрузках.

Унифицирующая идея Macintosh проявляется настолько сильно, что большинство других вариантов конструкции, описанных выше, либо находятся под ее влиянием, либо незаметны. Во всех программах предусмотрены графические интерфейсы. Интерфейса командной строки не существует вообще. Средства сценариев есть в наличии, однако они используются значительно реже, чем в Unix; многие Mac-программисты никогда их не изучают. Присущая MacOS метафора неотделимого GUI-интерфейса (организованная вокруг единственного главного событийного цикла) обусловила наличие слабого планировщика задач без приоритетности обслуживания. Слабый планировщик и работа всех MultiFinder-приложений в одном большом адресном пространстве означают, что использовать отдельные процессы или даже параллельные процессы вместо поочередного опроса непрактично.

Вместе с тем приложения MacOS не являются неизменно огромными монолитами. Системный код поддержки графического интерфейса, который частично реализован в ПЗУ, поставляемом с аппаратным обеспечением, а частично в совместно используемых библиотеках, обменивается данными с MacOS-программами посредством интерфейса событий, который с момента возникновения является весьма стабильным. Таким образом, конструкция данной операционной системы поддерживает относительно четкое обособление ядра приложения от GUI-интерфейса.

В MacOS также имеется мощная поддержка для изоляции метаданных приложений, таких как структуры меню, от кода ядра. Файлы данной операционной системы имеют как "ветвь данных" (data fork) (блок байтов в Unix-стиле, который содержит документ или программный код), так и "ветвь ресурсов" (resource fork) (набор определяемых пользователем атрибутов файла). Mac-приложения часто проектируются так для того, чтобы (например) используемые в приложениях изображения и звук хранились в ветви ресурса и чтобы их можно было бы модифицировать отдельно от кода приложения.

Система внутренних границ MacOS является очень непрочной. Имеется жесткое предположение о том, что есть только один пользователь, поэтому групп привилегий не существует. Многозадачность определяется как невытесняющая. Все MultiFinder-приложения запускаются в одном адресном пространстве, поэтому некорректный код какого-либо приложения способен разрушить любые данные, находящиеся за пределами низкоуровневого ядра операционной системы. Взломать систему безопасности MacOS-машин весьма просто. Данная операционная система избавлена от вирусных эпидемий главным образом потому, что очень немногие заинтересованы в ее взломе.

Mac-программисты стремятся разрабатывать приложения в противоположном относительно Unix-программирования направлении, т.е. это направление от интерфейса к ядру, а не

наоборот (некоторые последствия такого выбора будут рассмотрены в главе 20). Такой подход поощряется всей конструкцией MacOS.

Macintosh задумывалась как клиентская операционная система для нетехнических конечных пользователей, что предполагает очень низкую толерантность относительно сложности интерфейса. Разработчики в Mac-культуре достигли значительных успехов в проектировании простых интерфейсов.

Затраты пользователя, решившего стать разработчиком, при условии наличия компьютера Macintosh никогда не были высокими. Таким образом, несмотря на довольно сложные интерфейсы, в Mac-сообществе очень рано сложилась устойчивая культура энтузиастов. Наблюдается сильная традиция небольших инструментов, условно бесплатных программ и программного обеспечения, поддерживаемого пользователями.

Классическая система MacOS устаревает. Большинство имеющихся в ней средств импортируются в MacOS X, которая объединяет их с Unix-инфраструктурой, вышедшей из традиций университета в Беркли[27]. В то же время лидирующие Unix-системы, например Linux, начинают заимствовать у MacOS такие идеи, как использование атрибутов файлов (обобщение ветви ресурса).

3.2.3. OS/2

Операционная система OS/2 зародилась как опытный проект компании IBM, называвшийся ADOS (Advanced DOS), один из трех претендентов на роль DOS 4. В то время компании IBM и Microsoft формально сотрудничали при разработке

операционной системы следующего поколения для компьютеров PC. OS/2 версии 1.0 впервые вышла в 1987 году для компьютеров с процессорами 286-й серии, но не имела успеха. Версия 2.0 для процессоров 386 появилась в 1992 году, но к этому времени альянс IBM/Microsoft уже распался. Microsoft с системой Windows 3.0 двигалась в другом (более выгодном) направлении. OS/2 привлекала преданное меньшинство последователей, но так и не смогла привлечь критическую массу разработчиков и пользователей. На рынке настольных систем она оставалась третьей позади Macintosh до тех пор, пока не была отнесена к Java-инициэтиве IBM 1996 года. Последней была версия 4.0 в 1996 году. Ранние версии нашли свое применение во встроенных системах и в момент написания книги (середина 2003 года) продолжают работать во многих машинах автоматизированных справочных служб во всем мире.

Подобно Unix, OS/2 была создана с поддержкой вытесняющей многозадачности и не работала бы без блока MMU (ранние версии имитировали MMU с помощью сегментации памяти в 286 процессорах). В отличие от Unix, OS/2 никогда не создавалась для работы в качестве многопользовательской системы. Создание дочерних процессов было относительно недорогим, но межпроцессное взаимодействие было сложным и ненадежным. Поддержка сети первоначально сводилась к LAN-протоколам, однако в более поздних версиях был добавлен набор протоколов TCP/IP. В OS/2 не было программ, аналогичных системным службам Unix, поэтому данная система никогда не обеспечивала многофункциональную поддержку сети.

В данной операционной системе были как CLI-, так и GUI-интерфейс. Большинство положительных отзывов, касающихся OS/2, относились к ее рабочему столу, Workplace Shell (WPS). Часть этой технологии была лицензирована у разработчиков AmigaOS Workbench, революционного графического интерфейса настольных систем, который до 2003 года имел

верных почитателей в Европе[28]. Это одна из областей дизайна, где OS/2 приобрела такой потенциал, которого Unix, вероятно, еще не достигла. Оболочка WPS представляла собой четкую, мощную, объектно-ориентированную конструкцию с ясным режимом работы и хорошей расширяемостью. По прошествии нескольких лет она станет исходной моделью для Linux-проекта GNOME.

Конструкция WPS с иерархией классов была одной из унифицирующих идей операционной системы OS/2. Другой идеей была мультипроцессная обработка. OS/2-программисты использовали организацию параллельной обработки в большой степени как частичную замену IPC между равноправными процессами. Традиции создания взаимодействующих инструментов не развивались.

OS/2 имела внутренние границы, которые можно было бы ожидать в однопользовательской операционной системе. Работающие процессы были защищены друг от друга, пространство ядра было защищено от пользовательского пространства, но пользовательских групп привилегий не было. Это означало, что файловая система не была защищена от злонамеренного кода. Другим следствием было отсутствие аналога каталога "/home"; данные приложений были разбросаны по всей системе.

Еще одним следствием недостатка многопользовательских возможностей было то, что в пользовательском пространстве не могло быть осуществлено разграничение привилегий. Таким образом, разработчики были склонны доверять только коду ядра. Многие системные задачи, которые в Unix обрабатывались бы демонами пользовательского пространства, были нагромождены в ядре или WPS-оболочке, в результате чего обе эти подсистемы сильно разрастались.

OS/2 имела текстовый, а не двоичный режим (т.е. режим, в котором последовательность CR/LF читалась как один символ конца строки, в сравнении с режимом, в котором подобная интерпретация не осуществлялась), но другой структуры записи файлов не было. Описываемая система поддерживала атрибуты файлов, которые использовались для сохранения постоянства рабочего стола подобно Macintosh. Системные базы данных хранились главным образом в двоичных форматах.

Предпочтительным стилем пользовательского интерфейса постоянно оставалась оболочка WPS. Пользовательские интерфейсы часто были более эргономичными, чем в Windows, хотя и не достигали стандартов Macintosh (период наиболее активного применения OS/2 наблюдался относительно раньше). Подобно Unix и Windows, пользовательский интерфейс OS/2 был организован вокруг множества независимых групп окон для различных задач, вместо захвата рабочего стола действующим приложением.

Целевой аудиторией OS/2 были предприятия и нетехнические пользователи, в связи с чем предполагалась низкая толерантность относительно сложности интерфейса. Данная система использовалась как в качестве клиентской, так и в качестве файлового сервера и сервера печати.

В начале 90-х годов разработчики сообщества OS/2 начали переходить к Unix-подобной среде, эмулирующей POSIX-интерфейсы, которая называлась EMX. Перенесенное с Unix программное обеспечение начало регулярно появляться в OS/2 во второй половине 90-х годов прошлого века.

Система EMX была доступна для загрузки без ограничений и включала в себя коллекцию GNU-компиляторов и другие средства разработки с открытым исходным кодом. Компания IBM периодически предоставляла копии системной документации по инструментарию OS/2-разработчика, которая была доступна на многих BBS-системах и FTP-серверах. Вследствие этого размер FTP-архива, разработанного пользователями программного

обеспечения для OS/2 ("Hobbes"), к 1995 году превысил гигабайт. Жизнеспособная традиция небольших инструментов, исследовательского программирования и условно бесплатных программ развивалась и привлекала верных последователей в течение нескольких лет после того, как OS/2 была отправлена на свалку истории.

После выхода операционной системы Windows 95, сообщество OS/2, находясь под влиянием окружения Microsoft и будучи поддерживаемым IBM, стало все более интересоваться языком Java. После того как в начале 1998 года был опубликован исходный код браузера Netscape, направление миграции изменилось (достаточно неожиданно) в сторону Linux.

OS/2 представляет интерес как учебный пример того, как далеко может продвинуться конструкция многозадачной, но однопользовательской операционной системы. Большинство наблюдений в этом учебном примере применимы к другим операционным системам того же общего типа, в особенности AmigaOS[29], и GEM[30]. Очень многие материалы по OS/2, включая несколько хороших архивов, в 2003 году все еще оставались доступными в Web[31].

3.2.4. Windows NT

Windows NT (New Technology) — операционная система корпорации Microsoft для использования на мощных персональных компьютерах и серверах. Она поставляется в нескольких вариантах, которые в контексте данного обзора могут рассматриваться как одно целое. Все операционные системы Microsoft с момента упадка Windows ME в 2000 году основываются на технологии Windows NT. Windows 2000 была NT версии 5, а Windows XP (текущая версия в 2003 году) — NT 5.1. Windows NT генетически произошла от операционной системы VMS, с которой она имеет несколько общих важных характеристик.

Windows NT выросла путем приращений, а поэтому испытывает недостаток унифицирующей метафоры, соответствующей идее Unix о том, что "каждый объект является файлом", или идее рабочего стола в MacOS[32]. Поскольку основные технологии не связаны в небольшом наборе устойчивых центральных метафор, они устаревают каждые несколько лет. Каждое из поколений технологии — DOS (1981), Windows 3.1 (1992), Windows 95 (1995), Windows NT 4 (1996), Windows 2000 (2000), Windows XP (2002) и Windows Server 2003 (2003) — требует, чтобы разработчики по-новому повторно изучали фундаментальные принципы, при этом прежний путь объявляется устаревшим и более не поддерживается на достаточном уровне.

Имеются также другие последствия такого подхода.

• Средства GUI неудобно соседствуют со слабым, отжившим интерфейсом командной строки, унаследованным от DOS и VMS.

• Программирование сокетов не имеет унифицирующего объекта данных, аналогичного Unix-идее "каждый объект является файлом", поэтому мультипрограммирование (multiprogramming) и сетевые приложения, являющиеся простыми в Unix, предполагают наличие нескольких более фундаментальных концепций в Windows NT.

Windows NT имеет атрибуты файлов в некоторых типах ее файловых систем. Они используются ограниченным способом для реализации списков доступа в некоторых файловых системах и не очень сильно влияют на стиль разработки. Кроме того, данная система характеризуется тем, что запись текстовых и двоичных файлов осуществляется по-разному, и это приводит к определенным неудобствам (как NT, так и OS/2 унаследовали этот симптом от DOS).

Хотя в системе поддерживается вытесняющая многозадачность, создание дочерних процессов является затратным, правда, не таким, как в VMS, но (на уровне 0,1 с на создание подпроцесса) почти на порядок более затратными, чем в современных Unix-системах. Средства создания сценариев слабые, и операционная система вынуждает широко использовать двоичные форматы. В дополнение к ожидаемым последствиям, которые были рассмотрены выше, рассмотрим ряд менее очевидных.

• Большинство программ вообще невозможно связать в сценарий. Для взаимодействия друг с другом программы опираются на сложные, неустойчивые методы

удаленного вызова процедур (Remote Procedure Call — RPC) , которые являются причиной многочисленных ошибок.

• Общих инструментов вообще не существует. Документы и базы данных невозможно читать или редактировать без специализированных программ.

• С течением времени CLI-интерфейс все более игнорируется, поскольку все реже используется в среде. Проблемы, связанные со слабым CLI-интерфейсом, скорее усугубляются, нежели решаются. (Отчасти в Windows Server 2003 предпринята попытка изменить данную тенденцию.)

Системные и конфигурационные данные пользователей централизованы в главном системном реестре, а не разбросаны во многих скрытых пользовательских файлах конфигурации и системных файлах данных, как в Unix. Это также имеет несколько последствий для всей конструкции.

• Реестр делает систему полностью неортогональной. Одиночные сбои в приложениях могут повредить данные реестра, часто делая невозможным использование всей системы и вызывая необходимость переустановки.

• Феномен

сползания реестра (registry creep) : по мере роста реестра увеличивающиеся затраты на доступ замедляют работу всех программ.

NT-системы в Internet печально известны своей уязвимостью для "червей", вирусов, повреждениями и взломами всех видов. Для этого имеется множество причин, некоторые из них более существенные, чем другие. Наиболее фундаментальной причиной является то, что внутренние границы в NT являются чрезвычайно проницаемыми.

В Windows NT предусмотрены списки управления доступом, с помощью которых возможна реализация групп привилегий пользователей, но большое количество унаследованного кода игнорирует их, а операционная система допускает это для того, чтобы не повредить обратной совместимости. Отсутствуют средства управления безопасностью сообщений между GUI-клиентами[33], а их добавление также нарушило бы обратную совместимость.

Несмотря на то, что в NT используется MMU, в версиях данной системы после 3.5, исходя из соображений производительности, графический интерфейс системы встроен в то же адресное пространство, что и привилегированное ядро. Последние версии содержат в пространстве ядра даже Web-сервер в попытке достичь скорости, присущей Web-серверам на основе Unix.

Подобные бреши во внутренних границах производят синергетический эффект, делая действительную безопасность NT-систем фактически невозможной[34]. Если нарушитель может запустить код как практически любой пользователь (например, используя способность программы Outlook обрабатывать почтовые макросы), то данный код может

фальсифицировать сообщения через систему окон к любому другому запущенному приложению. Любое переполнение буфера или взлом в GUI-интерфейсе или Web-сервере также может быть использован для захвата контроля над всей системой.

Поскольку Windows должным образом не управляет контролем версий программных библиотек, данная система страдает от хронической конфигурационной проблемы, которая называется "DLL hell" (ад DLL). Данная проблема связана с тем, что при установке новых программ возможно случайное обновление (или напротив, запись более старых версий поверх новых) библиотек, от которых зависят существующие программы. Это относится как к системным библиотекам, поставляемым производителем, так и к библиотекам приложений: нередко приложения поставляются с определенными версиями системных библиотек, и в случае их отсутствия аварийно завершают работу[35].

К важнейшим свойствам Windows NT можно отнести предоставление данной системой достаточных средств для поддержки технологии Cygwin, которая представляет собой уровень совместимости, реализующий Unix как на уровне утилит, так и на уровне API-интерфейсов, с необыкновенно малым количеством компромиссов[36]. Cygwin позволяет C-программам использовать как Unix, так и Windows API-интерфейсы, а поэтому пользуется чрезвычайной популярностью у Unix-хакеров, вынужденных работать на Windows-системах.

Целевой аудиторией операционных систем Windows NT главным образом являются нетехнические конечные пользователи, которые характеризуются низкой толерантностью относительно сложности интерфейса. Windows NT используется как в роли клиентской, так и в роли серверной системы.

В начале своей истории корпорация Microsoft зависела от сторонних разработчиков, поставляющих приложения. Первоначально Microsoft публиковала полную документацию на Windows API и сохраняла невысокий уровень цен на средства разработки. Однако со временем и по мере исчезновения конкурентов, стратегия Microsoft сместилась в сторону поддержки собственных разработок, компания стала скрывать API-функции от внешнего мира, а средства разработки стали более дорогими. С выходом Windows 95 Microsoft стала требовать неразглашения соглашений в качестве условия приобретения профессиональных средств разработки.

Культура разработчиков-любителей и энтузиастов, которая выросла вокруг DOS и ранних версий Windows, была достаточно распространенной, чтобы оставаться самодостаточной вопреки нарастающим усилиям Microsoft, которая пыталась блокировать разработчиков (включая такие меры, как сертификационные программы, разработанные для того, чтобы объявить любителей вне закона). Условно бесплатные программы не исчезали, и политика Microsoft после 2000 года начала отчасти разворачиваться в обратном направлении под давлением рынка операционных систем с открытыми исходными кодами и языка Java. Однако Windows-интерфейсы для "профессионального" программирования с годами становились все более сложными, представляя увеличивающиеся барьеры для любительского (или серьезного) программирования.

В результате произошло предельно четкое разделение стилей разработки практикуемых любителями и профессиональными NT-разработчиками. Две эти группы только общаются. Тогда как любительская культура небольших инструментов и условно бесплатных программ весьма жива, профессиональные NT-проекты склонны к созданию огромных монолитов, даже более громоздких, чем программы, характерные для "элитных" операционных систем, наподобие VMS.

Unix-подобные средства оболочки, наборы команд и библиотечные API-интерфейсы доступны в Windows посредством библиотек сторонних разработчиков, включая UWIN, Interix и библиотеку с открытым исходным кодом Cygwin.

3.2.5. BeOS

Компания Be Inc. была основана в 1989 году как производитель аппаратного обеспечения в виде передовых многопроцессорных машин на основе микросхем PowerPC. Операционная система BeOS была попыткой компании Be повысить стоимость аппаратного обеспечения путем создания новой, сетевой модели операционной системы, учитывающей уроки Unix и MacOS, но не являющейся подобием одной из них. В результате появилась изящная, ясная и интересная конструкция с превосходной производительностью в предопределенной для нее роли мультимедийной платформы.

Унифицирующими идеями данной BeOS были "заполняющая параллельная обработка" (pervasive threading), мультимедийные потоки и файловая система, выполненная в виде базы данных. BeOS была спроектирована для минимизации задержек в ядре, что делало его применимым для обработки в реальном времени больших объемов таких данных, как аудио- и видео-потоки. "Пареллельные процессы" (threads) BeOS по существу были легковесными процессами в терминологии Unix, так как они поддерживали локальную память процесса и, следовательно, не обязательно совместно использовали все адресное пространство. Межпроцессный обмен данными посредством совместно используемой памяти был быстрым и эффективным.

BeOS придерживалась модели Unix в том, что не имела файловой структуры выше байтового уровня. Подобно MacOS, операционная система BeOS поддерживала и использовала атрибуты файлов. По сути, файловая система BeOS была базой данных с возможностью индексации по любому атрибуту.

Одним из элементов, заимствованных BeOS у Unix, была логичная конструкция внутренних границ. В описываемой системе полностью использовался блок MMU, и работающие процессы были надежно изолированы друг от друга. Несмотря на то, что BeOS представлялась как однопользовательская операционная система (без необходимости регистрации в системе), в ее файловой системе и во всем внутреннем устройстве поддерживались Unix-подобные группы привилегий. Они использовались для защиты важнейших системных файлов от воздействия ненадежного кода. В терминологии Unix пользователь во время загрузки регистрировался в системе как анонимный гость, а другим единственным "пользователем" был root. Полная многопользовательская функциональность потребовала бы небольших изменений в верхних уровнях системы и по сути была представлена утилитой BeLogin.

BeOS более стремилась к использованию двоичных форматов файлов и собственной базе данных, встроенной в файловую систему, чем к Unix-подобным текстовым форматам.

Предпочтительным стилем пользовательского интерфейса был GUI и он весьма склонялся к опыту MacOS в области дизайна интерфейсов. Вместе с тем также полностью поддерживались CLI-интерфейс и сценарии. Оболочкой командной строки была перенесенная с Unix программа

bash(1) , доминирующая Unix-оболочка с открытым исходным кодом, работающая посредством библиотеки совместимости POSIX. Благодаря такой конструкции, преобразование программного обеспечения Unix CLI было очень простым. В системе присутствовала инфраструктура для поддержки всего разнообразия сценариев, фильтров и служебных демонов, сопутствующих Unix-модели.

BeOS была предназначена для работы в качестве клиентской операционной системы,

специализирующейся на обработке мультимедийных данных (особенно обработке звука и видео) почти в реальном времени. Целевая аудитория данной системы включала в себя технических и деловых конечных пользователей, предполагающих умеренную толерантность к сложности интерфейса.

Препятствия на пути к разработке BeOS-приложений были небольшими. Несмотря на то, что операционная система была частной, средства разработки были недорогими, и доступ к полной документации не представлял сложности. Проект BeOS начался как часть усилий по освобождению от аппаратного обеспечения Intel с RISC-технологией, и после взрывного роста Internet продолжался исключительно как программный проект. Его стратеги изучили опыт периода формирования Linux в начале 90-х годов и были полностью осведомлены о значении крупной базы любительской разработки. Фактически они преуспели в привлечении верных последователей; в 2003 году не менее пяти отдельных проектов пытались возродить операционную систему BeOS в открытом исходном коде.

К сожалению, в отличие от технического дизайна, окружавшая BeOS бизнес- стратегия была не столь мудрой. Программное обеспечение BeOS первоначально было привязано к специализированному аппаратному обеспечению и продавалось только с неопределенными указаниями о целевых приложениях. Позднее (в 1998 году) операционная система BeOS была перенесена на общее аппаратное обеспечение PC, а мультимедийным приложениям было уделено более пристальное внимание, но система так и не привлекла критическую массу приложений или пользователей. Наконец, в 2001 году BeOS уступила комбинации антиконкурентного маневрирования Microsoft (судебный процесс продолжался в 2003 году) и конкуренции со стороны вариантов операционной системы Linux, адаптированных для обработки мультимедиа.

3.2.6. MVS

MVS (Multiple Virtual Storage) — ведущая операционная система IBM для мэйнфреймов корпорации. Ее происхождение связывают с OS/360, операционной системой IBM, появившейся в середине 60-х годов прошлого века. IBM планировала, что данная система будет использоваться клиентами на новых в то время компьютерных системах System/360. Потомки этого кода остаются основой сегодняшних операционных систем для мэйнфреймов IBM. Хотя код был почти полностью переписан, основная конструкция осталась почти совершенно нетронутой. Обратная совместимость поддерживается настолько тщательно, что приложения, написанные для OS/360, работают без модификаций в MVS для 64-битовых мэйнфреймов z/Series, появившихся на 3 архитектурных поколения позже.

Из всех рассматриваемых здесь операционных систем только MVS может считаться более старшей, чем Unix. Данная система также менее остальных подверглась влиянию идеи и технологии Unix и представляет надежнейшую конструкцию, противоположную последней. Унифицирующей идеей MVS является то, что вся работа формируется в виде пакета. Система разработана для наиболее эффективного использования машины для пакетной обработки больших объемов данных с минимальной необходимостью взаимодействия с пользователями.

Собственные MVS-терминалы (серии 3270) функционируют только в режиме блокировки. Пользователю предоставлен экран, который он заполняет, модифицируя локальную память терминала. Прерывание не происходит до тех пор, пока пользователь не нажмет клавишу отправки. Взаимодействие с помощью командной строки, подобное Unix-режиму непосредственного ввода данных с клавиатуры, невозможно.

Оснастка TSO, ближайший эквивалент интерактивной среды Unix, ограничена в собственных возможностях. Каждый пользователь TSO представлен остальной системе в виде условного пакетного задания. Данное средство является настолько дорогим, что круг его пользователей обычно ограничен программистами и обслуживающим персоналом. Обычно пользователи, которым необходимо только запускать приложения из терминала, почти никогда не используют TSO. Вместо этого они работают через мониторы транзакций, которые являются одним из видов многопользовательского сервера приложений, поддерживающего невытесняющую многозадачность и асинхронный ввод-вывод. В сущности, каждый вид монитора транзакций представляет собой специализированный дополнительный модуль разделения времени (отчасти подобный Web-серверу, выполняющему CGI-программу).

Другое следствие архитектуры, ориентированной на пакетную обработку, заключается в том, что создание подпроцессов является медленной операцией. В данной системе намеренно большая пропускная способность достигается ценой дорогостоящей установки (и связанной с этим задержки). Подобный подход хорошо соответствует пакетным операциям, но плохо сказывается на интерактивном отклике. Предсказуемым результатом является то, что сегодняшние пользователи TSO почти все время проводят в диалоговой интерактивной среде, ISPF. В редких случаях программисты делают что-либо внутри собственной среды TSO за исключением запуска экземпляра ISPF. Это устраняет издержки создания дочерних процессов ценой введения очень большой программы, которая выполняет все, кроме включения кофеварки.

Операционная система MVS использует аппаратный блок MMU. Процессы выполняются в отдельных адресных пространствах. Межпроцессный обмен данными осуществляется через совместно используемую память. В системе имеются средства для организации параллельной обработки (которая в MVS называется "созданием подзадач" (subtasking)), однако они используются незначительно, главным образом ввиду того, что данное средство легко доступно только из программ, написанных на ассемблере. Вместо этого типичное пакетное приложение представляет собой короткие серии вызовов тяжеловесных программ, связанных вместе с помощью JCL (Job Control Language — язык управления заданиями), обеспечивающим создание сценариев трудоемким и негибким способом. Программы в задании сообщаются посредством временных файлов. Фильтры и подобные им средства почти невозможно реализовать удобным способом.

Каждый файл имеет формат записи. Иногда формат подразумевается (например, предполагается, что встроенные в JCL файлы ввода имеют формат записи фиксированной длины (80 байт), унаследованный от перфокарт), но чаще он указывается в явном виде. Многие файлы системной конфигурации имеют текстовый формат, но файлы приложений обычно записываются в двоичных форматах, специфичных для определенного приложения. Некоторые общие инструменты для просмотра файлов развились из абсолютной необходимости, но до сих пор не являются простой для разрешения проблемой.

Безопасность файловой системы была поздним дополнением к первоначальной конструкции. Однако когда выяснилось, что безопасность необходима, IBM добавила соответствующие функции оригинальным способом: разработчики определили общий API-интерфейс функций безопасности, а затем все запросы на доступ к файлам перед обработкой направили через данный интерфейс. В результате существует по крайней мере три конкурирующих пакета обеспечения безопасности с различной философией дизайна, и все они весьма хороши, учитывая то, что известных взломов между 1980 и 2003 годами не было. Это многообразие позволяет при инсталляции выбрать пакет, который наилучшим образом подходит для локальной политики безопасности.

Сетевые средства также были добавлены с опозданием. В описываемой системе отсутствует понятие одного интерфейса для сетевых соединений и локальных файлов. Их программные интерфейсы разделены и полностью различны. Это действительно позволило набору

протоколов TCP/IP достаточно безболезненно вытеснить собственную модель сети IBM — SNA (Systems Network Architecture — системная сетевая архитектура), которая считалась предпочтительным сетевым протоколом. В 2003 году все еще можно было увидеть использование обеих архитектур в определенной инсталляции, однако SNA все же отмирает.

Любительское программирование для MVS почти отсутствует. И существует только внутри сообщества крупных предприятий, использующих данную операционную систему. Это связано не столько с затратами на сами инструменты, сколько со стоимостью среды. Когда предприятие вынуждено израсходовать несколько миллионов долларов на компьютерную систему, несколько сотен долларов, потраченных в месяц на компилятор, считаются почти мелкими расходами. Однако внутри данного сообщества существует процветающая культура свободно доступного программного обеспечения, главным образом средств программирования и системного администрирования. Первая группа компьютерных пользователей SHARE была основана пользователями IBM в 1955 году, и с тех пор устойчиво развивается.

При рассмотрении значительных архитектурных различий примечательным является тот факт, что MVS была первой операционной системой, имеющей стиль, отличный от System V, т.е. соответствовала единому стандарту Unix (это не столь заметно, чем то, что перенесенные Unix-программы имеют сильную склонность к нестабильной работе с проблемами символьного набора ASCII-EBCDIC). Запустить оболочку Unix из TSO вполне возможно; файловые системы Unix являются специально отформатированными группами данных MVS. Набор символов MVS Unix является специальной кодовой страницей EBCDIC с замененными символами новой строки и перевода строки. То, что в Unix представлено как перевод строки, в MVS выглядит как новая строка. Однако системные вызовы являются действительно системными вызовами, реализованными в ядре MVS.

Поскольку стоимость среды резко снижается, уже определилась небольшая, но растущая группа пользователей последней свободно распространяемой версии MVS (3.8, датированной 1979 годом). Эта система, как и все средства разработки, а также эмулятор для их запуска, доступны по цене компакт-диска[37].

MVS всегда предназначалась для работы в сопровождении. Как VMS и сама Unix, операционная система MVS предшествовала разделению клиент/сервер. Сложность интерфейса для пользователей сопровождения не только допустима, но и желательна в целях сокращения затрат дорогостоящих ресурсов на интерфейсы, а значит, выделения больших ресурсов для основной работы.

3.2.7. VM/CMS

VM/CMS —

другой пример операционной системы для мэйнфреймов. Ее вполне можно назвать "родственницей" системы Unix: их общим предком является система CTSS, созданная в Массачусетском технологическом институте приблизительно в 1963 году и работавшая на мэйнфрейме IBM 7094. Группа, разработавшая CTSS, позднее приступила к написанию Multics, прямого предка Unix. IBM учредила в Кембридже группу по созданию системы разделения времени для IBM 360/40, модифицированного компьютера 360-й серии со страничным (впервые для систем IBM) диспетчером MMU[38]. Программисты MIT и IBM впоследствии в течение многих лет продолжали взаимодействовать. Новая система приобрела пользовательский интерфейс, очень сходный с CTSS, укомплектованный оболочкой, которая называлась EXEC, а также большим запасом утилит аналогичных

используемым в Multics и позднее в Unix.

В другом смысле VM/CMS и Unix являлись видоизмененными "отражениями" друг друга. Унифицирующая идея системы, обеспеченная компонентом VM, воплощена в виртуальных машинах, которые выглядят как физические машины. Они поддерживают вытесняющую многозадачность и работают либо с однопользовательской операционной системой CMS, либо с полностью многозадачной операционной системой (обычно MVS, Linux или другой экземпляр самой VM). Виртуальные машины соответствуют Unix-процессам, демонам и эмуляторам, а обмен данными между ними осуществляется путем соединения виртуального карточного перфоратора одной машины с виртуальным считывателем перфокарт другой машины. В дополнение к этому, внутри системы обеспечивается многоуровневая инструментальная среда, которая называется CMS Pipelines (конвейеры CMS), непосредственно смоделированная с каналов Unix, но архитектурно расширенная для поддержки множества вводов и выводов.

В ситуации, когда обмен данными между виртуальными машинами явно не установлен, они полностью изолированы друг от друга. Данная операционная система характеризуется тем же высоким уровнем надежности, расширяемости и безопасности, что и MVS, а также имеет гораздо большую гибкость и проще в использовании. В дополнение к этому, VM-компонент, который обслуживается полностью обособленно, вовсе не обязательно должен доверять частям CMS, подобным ядру.

Несмотря на то, что CMS является операционной системой, ориентированной на структуры записи, эти записи, в сущности, эквивалентны строкам, используемым текстовыми инструментами в Unix. Базы данных значительно полнее интегрированы в CMS Pipelines, чем обычно в Unix, где большинство баз данных полностью отделены от операционной системы. В последние годы операционная система CMS была дополнена для полной поддержки единого стандарта Unix.

Стиль пользовательского интерфейса в CMS является интерактивным и диалоговым, весьма отличающимся от MVS, но похожим на пользовательские интерфейсы VMS и Unix. Интенсивно используется полноэкранный редактор XEDIT.

VM/CMS предшествовала разделению клиент/сервер и в настоящее время используется почти полностью как серверная операционная система с эмуляцией IBM-терминалов. До того как Windows стала полностью доминировать на рынке настольных систем, VM/CMS предоставляла службы обработки текстов и электронную почту как внутри IBM, так и между участками пользователей мэйнфреймов. Действительно, многие VM-системы были инсталлированы исключительно для запуска таких приложений благодаря доступной расширяемости VM (десятки тысяч пользователей).

Язык сценариев REXX поддерживает программирование в стиле, не отличающемся от shell, awk, Perl или Python. Следовательно, любительское программирование (особенно системными администраторами) является весьма важным в системе VM/CMS. Администраторы, располагающие недорогими ресурсами, часто предпочитают запускать действующую MVS в виртуальной машине, а не непосредственно на физической машине, для того чтобы CMS также была доступна и можно было использовать преимущества ее гибкости. (Существует ряд CMS-инструментов, предоставляющих доступ к файловым системам MVS.)

Прослеживаются поразительные параллели между историей VM/CMS внутри корпорации IBM и Unix внутри Digital Equipment Corporation (которая создавала компьютеры, где впервые работала Unix). Компании IBM потребовались годы, чтобы осознать стратегическую важность ее неофициальной системы разделения времени, в течение этого периода появилось сообщество программистов VM/CMS, поведение которого было весьма сходно с поведением раннего Unix-сообщества. Они обменивались идеями, открытиями в исследовании систем, а

главное — обменивались исходным кодом для утилит. Независимо от того, как часто IBM пыталась объявлять о смерти системы VM/CMS, сообщество, которое включало в себя собственно разработчиков системы MVS в IBM, настаивало на поддержании ее в рабочем состоянии. Операционная система VM/CMS прошла тот же путь от открытого исходного кода (де-факто) к закрытому и обратно, хотя этот процесс и не был столь явно выраженным, как в Unix.

Однако системе VM/CMS не хватает какого-либо реального аналога для языка C. Как VM, так и CMS были написаны на ассемблере и остаются такими и поныне. Единственным эквивалентом C были различные сокращенные версии языка PL/I, который использовался в IBM для системного программирования, однако клиентам компании не предоставлялся. Таким образом, данная система остается в ловушке ее собственной архитектурной линии, хотя она выросла и расширилась по мерее того, как архитектура 360 перерастала в серию 370, серию XA и наконец в современную серию z/Series.

С 2000 года IBM очень активно продвигает операционную систему VM/CMS на мэйнфреймах как способ одновременной поддержки тысяч виртуальных Linux-машин.

3.2.8. Linux

Операционная система Linux, созданная Линусом Торвальдсом в 1991 году, лидирует среди Unix-систем новой школы с открытым исходным кодом, появившихся в 1990 году (в их число также входит FreeBSD, NetBSD, OpenBSD и Darwin), и представляет направление конструирования, принятое данной группой в целом. Тенденции Linux могут быть приняты как типичные для всей группы.

Linux не включает в себя код из дерева исходных кодов первоначальной Unix, но данная система была сконструирована на основе Unix-стандартов и работает подобно Unix. В остальной части данной книги неоднократно подчеркивается преемственность между Unix и Linux. Эта преемственность чрезвычайно сильна, как в аспекте технологии, так и в отношении главных разработчиков, однако здесь подчеркиваются и некоторые направления Linux, которые демонстрируют отклонение от "классических" традиций Unix.

Многие разработчики и активисты Linux-сообщества стремятся к тому, чтобы данная операционная система заняла прочные позиции на рабочих столах конечных пользователей. Это делает целевую аудиторию Linux несколько шире, чем в случае какой- либо из Unix-систем старой школы, которые в основном были нацелены на рынки серверов и рабочих станций технических пользователей. Данное обстоятельство, конечно же, влияет на способ разработки программного обеспечения Linux-хакерами.

Наиболее очевидным новшеством является смена предпочтительных стилей интерфейса. Unix первоначально была разработана для использования на телетайпах и медленных печатающих терминалах. На протяжении большей части своей истории она была жестко связана с символьными видеотерминалами с недостатком либо графических возможностей, либо возможностей передачи цвета. Большинство Unix-программистов долго оставались прочно привязанными к командной строке, после того как большое количество пользовательских приложений перешло на графические GUI-интерфейсы в X Window, и конструкция как операционных систем Unix, так и приложений для них все еще отражает данный факт.

С другой стороны, пользователи и разработчики Linux привыкли учитывать характерную для нетехнических пользователей боязнь командной строки. Они перешли к созданию

GUI-интерфейсов и GUI-средств гораздо более интенсивно, чем это было в случае с Unix старой школы, или даже в случае современных частных Unix-систем. В меньшей, но все же значительной степени это справедливо и для других Unix-систем с открытым исходным кодом.

Желание склонить на свою сторону конечных пользователей также заставило Linux-разработчиков гораздо больше интересоваться проблемами простоты инсталляции и распространения программного обеспечения, чем это принято при разработке частных Unix-систем. В результате в Linux появились более развитые системы бинарных пакетов, чем какие-либо аналоги в частных Unix-системах. Данные системы имеют интерфейсы, спроектированные (в 2003 году) в более приемлемой для нетехнических пользователей манере.

Linux-сообщество стремится (более чем когда-либо) превратить свое программное обеспечение в некий универсальный канал связи между различными средами. Поэтому Linux предоставляет поддержку чтения и (нередко) записи форматов файловых систем и методов сетевого взаимодействия, характерных для других операционных систем. Linux также поддерживает возможность выбора операционной системы при начальной загрузке на одном и том же аппаратном обеспечении (мультизагрузку), а также программную эмуляцию данных систем внутри самой себя. Долгосрочной целью является поглощение; Linux эмулирует другие системы, а значит, может абсорбировать их[39].

В целях поглощения конкурентов и привлечения конечных пользователей Linux-разработчики перенимают конструкторские идеи из операционных систем, не относящихся к семейству Unix. Это настолько распространено, что традиционные Unix-системы выглядят довольно ограниченными. Примером могут послужить Linux-приложения, использующие для конфигурации .INI-файлы формата Windows; данная тема рассматривается в главе 10. Внедрение в ядро 2.5 Linux расширенных атрибутов файлов, которые среди прочего можно использовать для эмуляции семантики ветви ресурса в Macintosh, — наиболее яркий пример на момент написания данной книги.

В тот день, когда Linux выдаст Mac-сообщение о невозможности открыть файл ввиду отсутствия соответствующего приложения, Linux перестанет быть Unix. Дуг Макилрой.

Остальные частные Unix-системы (такие как Solaris, HP-UX, AIX и другие) разрабатываются как большие продукты для больших IT-бюджетов. Их рыночная ниша поддерживает конструкции, оптимизированные под максимальную мощность на высококлассном, инновационном аппаратном обеспечении. Поскольку частично Linux связана со средой энтузиастов PC, особое значение в данной системе уделяется выполнению большего количества задач при меньших затратах. Частные Unix-системы настраиваются на многопроцессорные и кластерные операции на низкопроизводительном, маломощном аппаратном обеспечении. А основные разработчики Linux-систем открыто выбрали неприятие большей сложности и затрат на малопроизводительных машинах ради незначительного прироста производительности высококлассной аппаратуры.

Несомненно, значительная часть сообщества пользователей Linux понимает, что перед нами стоит задача извлечь максимальную пользу из аппаратуры, настолько же технически устаревшей сегодня, насколько в 1969 году устаревшим был компьютер Кена Томпсона PDP-7. Как следствие, Linux-разработчики вынуждены оставлять приложения скудными и неприятными, с чем их коллеги в частных Unix-системах не сталкиваются.

Эти тенденции, конечно же, скажутся на будущем Unix в целом. Данная тема рассматривается в главе 20.

3.3. Все повторяется

В данной главе для сравнения были выбраны системы разделения времени, которые либо в настоящее время, либо в прошлом составляли конкуренцию Unix. Достойных кандидатов не много. Большинство подобных систем (Multics, ITS, DTSS, TOPS-10, TOPS-20, MTS, GCOS, MPE и, возможно, десяток других) исчезли так давно, что почти стерлись из коллективной памяти компьютерной отрасли. Отмирают системы VMS и OS/2, MacOS поглощается Unix-производными. Операционные системы MVS и VM/CMS были ограничены одной частной линейкой мэйнфреймов. Только Microsoft Windows остается жизнеспособным конкурентом, независимым от традиций Unix.

Преимущества Unix рассматривались в главе 1, и они, несомненно, составляют некоторую часть объяснения. Однако, считаем, более полезным будет обсуждение другого очень важного аспекта данной проблемы: какие недостатки конкурентов Unix оставили их в проигрыше?

Наиболее очевидной общей проблемой является неспособность к переносу на другие платформы. Большинство конкурентов Unix, появившихся до 1980 года, были привязаны к какой-либо одной аппаратной платформе и исчезли вместе с ней. Единственной причиной того, что VMS просуществовала достаточно долго для того, чтобы рассматриваться здесь в качестве учебного примера, является то, что она была успешно перенесена с ее первоначального аппаратного обеспечения VAX на процессор Alpha (а в 2003 году переносился с Alpha на Itanium). MacOS была успешно перенесена с микросхем Motorola 68000 на PowerPC в конце 80-х годов прошлого века. Операционная система Microsoft Windows избежала данной проблемы, оказавшись в нужном месте, когда стремление превратить программное обеспечение в продукт массового потребления привело к заполнению рынка универсальными компьютерами монокультуры PC.

С 1980 года все более проявляется другой недостаток, характерный для различных операционных систем, которые либо были уничтожены Unix, либо просуществовали менее продолжительный период времени: неспособность обеспечить изящную поддержку сети.

В мире распространяющихся сетей даже операционная система, спроектированная для однопользовательской работы, нуждается в многопользовательских средствах (множество групп привилегий), поскольку без таких средств любая сетевая транзакция, которая способна обманным путем вынудить пользователя запустить злонамеренный код, может разрушить всю систему (макровирусы в Windows являются только верхушкой этого айсберга). Без мощной многозадачности способность операционной системы одновременно обрабатывать сетевой трафик и выполнять пользовательские программы будет ослаблена. Операционная система также должна обладать эффективным IPC-механизмом, для того чтобы ее сетевые программы могли обмениваться данными друг с другом и другими приоритетными пользовательскими приложениями.

Windows, имея серьезные недостатки в данных областях, избегает упадка только благодаря тому, что она получила монопольное положение еще до того, как сетевое взаимодействие стало действительно важным, а также благодаря множеству пользователей, которые в силу определенных условий вынуждены принимать как должное аварийные отказы и бреши в системе безопасности, количество которых шокирует. Данную ситуацию нельзя назвать стабильной. Сторонники Linux успешно ее использовали (в 2003 году), для того чтобы проникнуть на рынок серверных операционных систем.

Приблизительно в 1980 году во время расцвета персональных компьютеров проектировщики операционных систем отклонили Unix и традиционную технологию разделения времени как

тяжеловесную, Громоздкую и несоответствующую новым условиям, определяемым средой однопользовательских персональных машин. Вместе с тем GUI-интерфейсы требовали внедрения многозадачности, для того чтобы справляться с исполняемыми потоками, связанными с различными окнами и их элементами управления. Направленность на клиентские операционные системы была настолько сильной, что серверные операционные системы были отклонены подобно реликтам на паровой тяге из прошлой эпохи.

Но, как отметили разработчики BeOS, требования распространяющихся сетей невозможно удовлетворить без реализации какой-либо технологии, весьма близкой к общецелевому разделению времени. Однопользовательские клиентские операционные системы не способны преуспеть в Internet-мире.

Данная проблема привела к новому объединению клиентских и серверных операционных систем. Первые (до появления Internet) попытки равноправного сетевого взаимодействия через локальные сети в конце 80-х годов начали выявлять неадекватность конструкторской модели клиентских операционных систем. Для совместного использования данных в сети необходимы точки рандеву (rendezvous points) для этих данных. Следовательно, работа без серверов невозможна. В то же время опыт клиентских операционных систем Macintosh и Windows поднял уровень минимальных пользовательских навыков, которой устроил бы потребителей.

С не-Unix моделями разделения времени, фактически исчезнувшими к 1990 году, существовало не много возможных решений, которые проектировщики клиентских операционных систем могли противопоставить данной проблеме. Они могли выделить Unix (как это было в MacOS X), изобрести заново приблизительно эквивалентные функции по частям (как это сделано в Windows), или попытаться заново создать весь мир (как попыталась и не смогла BeOS). Однако тем временем в Unix-системах с открытым исходным кодом росли клиентские возможности по использованию GUI-интерфейсов, а также возможности работы на недорогих персональных машинах.

Данные факторы оказались, однако, не настолько уравновешенными, как это может показаться из приведенного выше описания. Модернизация таких функций серверных операционных систем, как множество классов привилегий и полная многозадачность, в клиентской операционной системе представляется очень трудной задачей, и, весьма вероятно, нарушит совместимость с предыдущими версиями клиента, а также приведет к многочисленным проблемам безопасности. С другой стороны, проблем связанных с внедрением GUI-интерфейса в серверную операционную систему, вполне можно избежать при условии наличия высококвалифицированного персонала и недорогих аппаратных ресурсов. Как и в строительстве, восстановить надстройку на поверхности прочного фундамента проще, чем заменить фундамент без разрушения надстройки.

Кроме наличия собственных архитектурных преимуществ серверной операционной системы, Unix всегда скептически относилась к целевой аудитории. Разработчики и создатели никогда не допускали, что знают все потенциально возможные способы применения системы.

Следовательно, конструкция операционной системы Unix проявила гораздо большие возможности по модернизации в качестве клиента, чем любая из клиентских операционных систем-конкурентов продемонстрировала возможности по модернизации в качестве сервера. Хотя возрождению Unix в 90-х годах прошлого века способствовало и множество других факторов, как технологических, так и экономических, но именно вышеупомянутые возможности выходят на первый план при обсуждении стиля проектирования операционных систем.

Часть II

Проектирование

4

Модульность: четкость и простота

Есть два способа конструирования программного обеспечения. Один из них заключается в том, чтобы создавать такие простые программы, в которых нет недостатков; другой — в том, чтобы создавать программы настолько сложные, чтобы в них не было очевидных недостатков. Первый метод значительно труднее.

The Emperor's Old Clothes, CACM, февраль 1981 -Ч. А. Р. Хоар (C. A. R. Hoare)

Существует естественная иерархия методов разделения кода на блоки, которая развивалась по мере того, как программистам приходилось управлять возрастающими уровнями сложности. Вначале все программы представляли собой огромную глыбу машинного кода. Ранние процедурные языки внесли понятие разделения на подпрограммы. Затем были изобретены служебные библиотеки для совместного использования общих полезных функций между множеством программ. После этого были разработаны отдельные адресные пространства и взаимодействующие процессы. В настоящее время не вызывает удивления тот факт, что программные системы распределяются среди множества узлов, разделенных тысячами миль сетевого кабеля.

Ранние разработчики Unix были в числе первопроходцев модульности программного обеспечения. До их прихода правило модульности было теорией компьютерной науки, но не инженерной практикой. В книге

"Design Rules" [2], радикальном учении об экономических основах модульности в инженерном дизайне, авторы используют развитие компьютерной индустрии в качестве учебного примера и доказывают, что сообщество Unix фактически было первым в систематическом применении модульной декомпозиции в производстве программного обеспечения в отличие от аппаратного обеспечения. Модульность аппаратного обеспечения, несомненно, была одной из основ инженерии с момента принятия стандартной винтовой резьбы в конце 19-го столетия.

Здесь следует акцентировать внимание на правиле модульности: единственным способом создания сложного программного обеспечения, которое будет надежно работать, является его построение из простых модулей, соединенных четкими интерфейсами, с тем чтобы большинство проблем были локальными, а также была большая вероятность наладки или оптимизации какой-либо его части без нарушения конструкции в целом.

Традиция заботиться о модульности и уделять пристальное внимание проблемам ортогональности и компактности до сих пор среди Unix-программистов проявляется гораздо ярче, чем в любой другой среде.

Программисты ранней Unix стали мастерами модульности, поскольку были вынуждены сделать это. Операционная система является одним из наиболее сложных блоков кода. Если

она не структурирована должным образом, то она развалится. Было несколько неудачных попыток построения Unix. Можно было бы отнести это на счет раннего (еще без структур) C, однако в основном это происходило ввиду того, что операционная система была слишком сложна в написании. Прежде чем мы смогли "обуздать" эту сложность, нам потребовалось усовершенствование инструментов (таких как структуры C)и хорошая практика их применения (например, правила Роба Пайка для программирования). Кен Томпсон.

Первые Unix-хакеры решали данную проблему различными способами. В 1970 году вызовы функций в языках программирования были очень затратными либо ввиду запутанной семантики вызовов (PL/1, Algol), либо из-за того, что компилятор был оптимизирован для других целей, например, для увеличения скорости внутренних циклов за счет времени вызова. Таким образом, код часто писали в виде больших блоков. Кен и несколько других первых Unix-разработчиков знали, что модульность является хорошей идеей, однако они помнили о PL/1 и неохотно писали небольшие функции, чтобы не снизить производительность.

Деннис Ритчи поддерживал модульность, повторяя всем без исключения, что вызовы функций в C крайне, крайне малозатратны. Все начали писать небольшие функции и компоновать программы из модулей. Через несколько лет выяснилось, что вызовы функций оставались дорогими на PDP-11, а код VAX часто тратил 50% времени на обработку инструкции CALLS. Деннис солгал нам! Однако было слишком поздно; все мы попались на удочку... Стив Джонсон.

Всех сегодняшних программистов, пишущих для Unix или других операционных систем, учат создавать модули внутри программ на уровне подпрограмм. Некоторые учатся делать это на уровне модулей или абстрактных типов данных и называют это "хорошим программированием". Сторонники использования моделей проектирования прилагают благородные усилия для повышения уровня разработки и создают все новые успешные абстрактные модели, которые можно применять для организации крупномасштабной структуры программ.

Здесь авторы не предпринимают попытки слишком подробно осветить все вопросы, связанные с модульностью внутри программ: во-первых, потому, что данная тема сама по себе является предметом целого тома (или нескольких томов), и, во-вторых, ввиду того, что настоящая книга посвящена искусству

Unix -программирования.

Здесь более конкретно рассматриваются уроки Unix-традиции, связанные с правилом модульности. В данной главе рассматривается разделение внутри процесса. Далее, в главе 7, будут рассматриваться условия, при которых следует разделять программы на множество взаимодействующих процессов, а также более специфические методики для осуществления такого разделения.

4.1. Инкапсуляция и оптимальный размер модуля

Первым и наиболее важным качеством модульного кода является

инкапсуляция . Правильно инкапсулированные модули не открывают свое внутренне устройство друг другу. Они не обращаются к центральной части реализации друг друга, кроме того, они не используют глобальные данные беспорядочно. Они осуществляют связь друг с другом посредством программных интерфейсов приложений (API) — компактных, четких

наборов вызовов процедур и структур данных. Именно об этом гласит правило модульности.

API-интерфейсам между модулями отведена двойная роль. На уровне реализации они функционируют как заслонка между модулями, которая предотвращает воздействие внутренних данных модулей на соседние модули. На уровне проектирования именно API-интерфейсы (а не описание структур данных между ними) действительно определяют структуру программ.

Хороший способ проверить правильность проектирования API-интерфейса — определить, будет ли ясен смысл, если попытаться описать API обычным человеческим языком (без демонстрации фрагментов исходного кода)? Весьма целесообразно выработать привычку писать неформальные описания для API-интерфейсов до их кодирования. Более того, некоторые из наиболее способных разработчиков начинают с определения интерфейсов и написания кратких комментариев для них, а затем пишут код, поскольку процесс написания комментариев разъясняет задачи, возлагаемые на код. Подобные описания помогают организовать мышление разработчика, а также создают полезные комментарии для модулей, и в конечном итоге их можно включить в справочный документ для будущих читателей кода.

Чем дальше проводится декомпозиция модулей, тем меньше становятся блоки и более важными определения API-интерфейсов, Сокращается глобальная сложность и, как следствие, уязвимость относительно ошибок. С 70-х годов прошлого века общепринятым правилом (описанном в статьях, подобных [61]) стало проектирование программных систем в виде иерархий вложенных модулей, с сохранением минимального размера модулей на каждом уровне.

Возможно, однако, что подобное разбиение на модули будет слишком строгим и приведет к появлению очень маленьких модулей. Доказано [33], что при построении диаграммы плотности дефектов относительно размера модуля, кривая становится U-образной, а ее лучи направлены вверх (см. рис. 4.1). Очень большие и очень малые модули обычно служат причиной возникновения гораздо большего количества ошибок, чем модули среднего размера. Другим способом рассмотрения тех же данных является построение диаграммы количества строк кода в модуле относительно общего количества ошибок. Кривая в таком случае подобна логарифмической кривой до зоны "наилучшего восприятия", где она сглаживается (соответственно с минимумом кривой плотности дефектов), и после которой она направляется вверх (как квадрат числа, соответствующего количеству строк кода, т.е. наблюдается то, что, согласно закону Брукса[40], можно интуитивно ожидать для всей кривой).

Рис. 4.1. Качественная диаграмма количества и плотности дефектов относительно размера модуля

Этот неожиданный рост количества ошибок при малых размерах модулей устойчиво наблюдается в широком диапазоне систем, реализованных на различных языках программирования. Хаттон (Hatton) предложил модель, связывающую данную нелинейную зависимость с объемом краткосрочной человеческой памяти[41]. Другая интерпретация данной нелинейной зависимости состоит в том, что при малых размерах элемента модуля возрастающая сложность интерфейсов становится доминирующим фактором. Трудно читать код, поскольку необходимо понять все еще до того, как станет возможным понять что-либо. В главе 7 рассматриваются более сложные формы разделения программ. В них также сложность интерфейсных протоколов начинает доминировать на фоне общей сложности системы по мере уменьшения размеров процессов

В нематематических понятиях эмпирические результаты Хаттона предполагают точку наилучшего восприятия между 200 и 400 логических строк кода, где сведена к минимуму возможная плотность дефектов, а все остальные факторы (такие как профессионализм

программиста) равны. Размер не зависит от используемого языка программирования. Это замечание весьма усиливает приводимый в данной книге совет о программировании с использованием наиболее мощных из доступных языков и инструментов. Не следует однако принимать данные числа слишком буквально. Методы подсчета строк кода в значительной степени различаются в зависимости от того, как аналитик определяет логическую строку, а также от других условий (например, от того, отбрасываются ли комментарии). Сам Хаттон в качестве приближенного подсчета советует использовать двукратное преобразование между логическими и физическими строками, рекомендуя оптимальный диапазон в 400–800 физических строк кода.

4.2. Компактность и ортогональность

Код не является единственным объектом, который имеет оптимальный размер своего элемента. Языки программирования и API-интерфейсы (например, библиотечных или системных вызовов) сталкиваются с теми же ограничениями человеческого восприятия, которые создают U-образную кривую Хаттона.

Следовательно, Unix-программисты научились весьма тщательно продумывать при проектировании API-интерфейсов, наборов команд, протоколов и других способов повышения эффективности два другие свойства:

компактность и

ортогональность .

4.2.1. Компактность

Компактность — свойство, которое позволяет конструкции "поместиться" в памяти человека. Для того чтобы проверить компактность, можно использовать хороший практический тест: необходимо определить, нуждается ли обычно опытный пользователь в руководстве для работы с данной программой. Если нет, то конструкция (или, по крайней мере, ее подмножество, которое охватывает обычное использование) является компактной.

Компактные программные средства обладают всеми достоинствами удобных физических инструментов. Их приятно использовать, они не препятствуют работе, позволяют повысить продуктивность.

Компактность не является синонимом "слабости". Конструкция может обладать большой мощностью и гибкостью, оставаясь при этом компактной, если она построена на абстракциях, которые просты в осмыслении и хорошо взаимосвязаны. Компактность также не эквивалентна "простоте обучения". Некоторые компактные конструкции являются весьма сложными для понимания до тех пор, пока пользователь не овладеет сложной основополагающей концептуальной моделью настолько, что его мировоззрение изменится, благодаря чему компактность

станет для него простой. Для многих классическим примером такой конструкции является язык Lisp.

Компактный не означает также "малый". Если четко спроектированная система является

предсказуемой и "очевидной" для опытного пользователя, то она может иметь довольно много частей. Кен Арнольд.

Очень немногие программные конструкции являются компактными в абсолютном смысле, однако многие являются компактными в более широком смысле. Они имеют компактный рабочий набор, подмножество возможностей, подходящее

для решения 80-ти или более процентов тех задач, для которых их обычно используют опытные пользователи. На практике для таких конструкций обычно требуется справочная карта или памятка, но не руководство. Подобные конструкции называются

полукомпактными в противоположность

строго компактным конструкциям.

Данная идея, возможно, лучше иллюстрируется с помощью примеров. API системных вызовов Unix является полукомпактным, однако стандартная библиотека C некомпактна в любом смысле. Тогда как Unix-программисты в своей памяти без затруднений содержат подмножество системных вызовов, достаточное для большей части прикладного программирования (операции с файловой системой, сигналы и управление процессами), библиотека C в современных Unix-системах включает в себя много сотен входных точек, например, математические функции, которые "не поместятся" в памяти одного программиста.

Статья

"The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" [54] является одним из основных документов по когнитивной психологии (и к тому же особой причиной того, что в Соединенных Штатах местные телефонные номера состоят из семи цифр). Статья показывает, что количество дискретных блоков информации, которое человек способен удерживать в краткосрочной памяти, равно семи плюс-минус два. Из этого следует хорошее практическое правило для оценки компактности API-интерфейсов: необходимо ли программисту помнить более семи входных точек? Любую более крупную конструкцию едва ли можно назвать компактной.

Среди инструментальных средств Unix компактной является утилита

make(1) , а

autoconf(1) и

automake(1) компактными не являются. Что касается языков разметки: HTML — полукомпактный язык, а DocBook (язык разметки документов, который будет рассматриваться в главе 18) таковым не является. Макросы

man(7) компактны, а разметка

troff(1) — нет.

Среди универсальных языков программирования C и Python являются полукомпактными, а Perl, Java, Emacs Lisp и shell — нет (особенно ввиду того, что серьезное shell-программирование требует от разработчика знаний нескольких других средств, таких как

sed(1) и

awk(1)). C++ — антикомпактный язык — его разработчик признал, что вряд ли какой-либо

один программист когда-нибудь поймет его полностью.

Некоторые конструкции, не являющиеся компактными, обладают такой внутренней избыточностью функций, что отдельные программисты с помощью рабочего подмножества языка создают для себя компактные диалекты, достаточные для решения 80% распространенных задач. Например, язык Perl характеризуется подобным типом псевдокомпактности. Такие конструкции имеют встроенные ловушки. В ситуации, когда два программиста пытаются обмениваться информацией по проекту, они могут обнаружить, что различия в их рабочих подмножествах являются значительными препятствиями для понимания и модификации кода.

В то же время некомпактные конструкции автоматически не являются безнадежными или плохими. Некоторые предметные области просто являются слишком сложными, для того чтобы компактный дизайн мог их охватить. Иногда необходимо пожертвовать компактностью в обмен на некоторые другие преимущества, такие как большая мощность и диапазон. Разметка Troff, а также API BSD-сокетов являются хорошими примерами таких конструкций. Компактность как преимущество подчеркивается здесь не для того, чтобы заставить трактовать ее как абсолютное требование, а для того, чтобы научить читателей поступать подобно Unix-программистам: должным образом ценить компактность, по возможности использовать ее в конструкциях и не отбрасывать ее по неосторожности.

4.2.2. Ортогональность

Ортогональность является одним из наиболее важных свойств, которое позволяет сделать даже сложные конструкции компактными. В исключительно ортогональных конструкциях операции не имеют побочных эффектов. Каждое действие (API-вызов, запуск макроса или операция языка) изменяет только один объект, не оказывая влияния на остальные. Существует один и только один способ для изменения каждого свойства любой управляемой системы.

Монитор компьютера имеет ортогональное управление. Яркость можно изменить независимо от уровня контрастности, а управление цветовым балансом не зависит от них обоих (если монитор имеет данную функцию). Представим, насколько более сложно было бы настраивать монитор, в котором регулятор яркости влиял бы на цветовой баланс: пришлось бы корректировать настройки цветового баланса каждый раз после изменения яркости. Еще хуже, если бы управление контрастностью также влияло на цветовой баланс. Пришлось бы манипулировать обоими регуляторами совершенно точно и одновременно, для того чтобы изменить по отдельности контраст или цветовой баланс при сохранении другого параметра постоянным.

Слишком многие программные конструкции являются неортогональными. Один общий класс ошибок проектирования, например, возникает в коде, который предназначен для считывания и преобразования данных из одного (исходного) формата в другой (целевой) формат. Проектировщик, который считает, что исходный формат всегда хранится в файле на диске, может написать функцию преобразования для открытия и чтения данных из именованного файла. Как правило, выполненный ранее ввод может также быть любым дескриптором файла. Если бы программа преобразования была спроектирована ортогонально, например, без побочных эффектов открытия файла, то она могла бы облегчить работу в дальнейшем, когда понадобилось бы преобразовать поток данных, поступающий из стандартного ввода, сетевого сокета или любого другого источника.

Совет Дуга Макилроя "решать одну задачу хорошо" обычно рассматривается в контексте

простоты. Однако он также неявно и, по крайней мере, в той же степени касается ортогональности.

В главе 9 рассматривается программа

ascii , которая печатает синонимы для названий ASCII-символов, включая шестнадцатеричные, восьмеричные и двоичные значения. Побочный эффект программы заключается в том, что она может служить в качестве быстрого конвертера для чисел в диапазоне 0-255. Это второе ее применение не является нарушением ортогональности, поскольку все функции, поддерживающие его, необходимы для реализации основной функции; они не усложняют документирование или поддержку программы.

Проблемы неортогональности возникают, когда побочные эффекты усложняют ментальную модель программиста или пользователя и забываются, что приводит к неприемлемым и даже фатальным результатам. Даже если побочные эффекты не забыты, часто для их подавления приходится выполнять дополнительную работу.

Превосходное обсуждение ортогональности и способов ее достижения приведено в книге

"The Pragmatic Programmer" [37]. Как указывают ее авторы, ортогональность сокращает время тестирования и разработки, поскольку проверка кода, который не вызывает побочных эффектов и не зависит от побочных эффектов другого кода, упрощается, и следовательно уменьшается количество тестовых комбинаций. Если ортогональный код работает неверно, то его просто заменить другим без нарушения остальной части системы. Наконец, ортогональный код является более простым для документирования и повторного использования.

Идея

рефакторинга (refactoring), которая впервые возникла как явная идея школы "экстремального программирования" (Extreme Programming), тесно связана с ортогональностью. Рефакторинг кода означает изменение его структуры и организации без изменения его видимого поведения. Инженеры программной индустрии, несомненно, решают эту проблему с момента возникновения отрасли, однако определение данной практики и идентификация главного набора методик рефакторинга способствовало решению данного вопроса. Поскольку все это хорошо согласуется с основными концепциями проектирования Unix, Unix-разработчики быстро переняли терминологию и идеи рефакторинга[42].

Основные API-интерфейсы Unix были спроектированы с учетом ортогональности не идеально, но вполне успешно. Например, возможность открытия файла для записи без его блокировки для остальных пользователей принимается как должное; не все операционные системы настолько "обходительны". Системные сигналы старой школы (System III) были неортогональными, поскольку получение сигнала имело побочный эффект — происходила переустановка обработчика сигналов в стандартное значение и отключение при получении сигнала. Существуют крупные неортогональные фрагменты, такие как API BSD-сокетов, и

очень крупные, такие как графические библиотеки системы X Window.

Однако в целом API-интерфейс Unix является хорошим примером ортогональности: иначе, он не только не был бы, но и

не мог бы широко имитироваться библиотеками C в других операционных системах. Это также является причиной того, что изучение Unix API окупается, даже для программистов, не работающих с Unix, поскольку они усваивают уроки ортогональности.

4.2.3. Правило SPOT

В книге

"The Pragmatic Programmer" формулируется правило для одного частного вида ортогональности, который является особенно важным. Это правило "не повторяйтесь": внутри системы каждый блок знаний должен иметь

единственное, недвусмысленное и надежное представление. В данной книге предпочтение отдано совету Брайана Кернигана называть данное правило SPOT-правилом (SPOT, или Single Point Of Truth, — единственная точка истины).

Повторение ведет к противоречивости и созданию кода, который незаметно разрушается, поскольку изменяются только некоторые повторения, когда необходимо изменить

все. Часто это также означает, что организация кода не была продумана должным образом.

Константы, таблицы и метаданные следует объявлять и инициализировать

только один раз , а затем импортировать. Всякое дублирование кода является опасным знаком. Сложность приводит к затратам; не следует оплачивать ее дважды.

Нередко имеется возможность удалить дублирование кода путем

рефакторинга , т.е. с помощью изменения организации кода без модификации основных алгоритмов. Иногда возникает дублирование данных, в таком случае следует ответить на несколько важных вопросов.

• Если дублирование данных существует в разрабатываемом коде ввиду необходимости иметь два различных представления в двух разных местах, то возможно ли написать функцию, средство или генератор кода для создания одного представления из другого или обоих из общего источника?

• Если документация дублирует данные из кода, то можно ли создать фрагменты документации из фрагментов кода или наоборот, или и то, и другое из общего представления более высокого уровня?

• Если файлы заголовков и объявления интерфейсов дублируют сведения в реализации кода, то существует ли способ создания файлов заголовков и объявлений интерфейсов из данного кода?

Для структур данных существует аналог SPOT-правила: "нет лишнего — нет путаницы". "Нет лишнего" означает, что структура данных (модель) должна быть минимальной, например, не следует делать ее настолько общей, чтобы она могла представлять ситуации, возникновение которых невозможно. "Нет путаницы" означает, что положения, которые должны быть обособлены в реальной проблеме, также должны быть обособлены в модели. Коротко говоря, SPOT-правило поддерживает поиск структуры данных, состояния которой имеют однозначное соответствие с состояниями реальной системы, которая будет моделироваться.

Авторы могут добавить некоторые собственные следствия SPOT-правила в контексте Unix-традиций.

• Если данные дублируются из-за кэширования промежуточных результатов некоторых вычислений или поиска, то следует внимательно проанализировать, не является ли это преждевременной оптимизацией. Устаревшие данные кэша (и уровни кода, необходимые для

поддержки синхронизации кэша) являются "неиссякаемым" источником ошибок[43] и даже способны снизить общую производительность, если (как часто случается) издержки управления кэшем превышают ожидания разработчика.

• Если наблюдается большое количество повторов шаблонного кода, то возможно ли создать их все из одного представления более высокого уровня, изменяя некоторые параметры для создания различных вариантов?

Теперь модель должна быть очевидной.

В мире Unix SPOT-правило редко проявляется как явная унифицирующая идея, однако, интенсивное использование генераторов кода для реализации специфических

видов SPOT является весьма большой частью традиции. Данные методики рассматриваются в главе 9.

4.2.4. Компактность и единый жесткий центр

Одним неочевидным, но мощным способом поддержать компактность в конструкции является ее организация вокруг устойчивого основного алгоритма, определяющего ясное формальное определение проблемы, избегая эвристики и ухищрений.

Формализация часто радикально проясняет задачу. Программисту не достаточно определить, что части поставленной перед ним задачи попадают в стандартные категории компьютерной науки — поиск и быстрая сортировка. Наилучшие результаты достигаются в том случае, когда можно формализовать суть задачи и сконструировать ясную модель работы. Вовсе не обязательно, чтобы конечные пользователи поняли данную модель. Само существование унифицирующей основы обеспечит ощущение комфорта, когда работа не затруднена вопросами типа "а зачем они сделали это", которые так распространены при использовании универсальных программ. Дуг Макилрой.

В этом заключается сила традиции Unix, но, к сожалению, это часто упускается из вида. Многие из ее эффективных инструментальных средств представляют собой тонкие упаковщики вокруг непосредственного преобразования некоторого единого мощного алгоритма.

Вероятно, наиболее ясным примером таких средств является программа

diff(1) , средство Unix для составления списка различий между связанными файлами. Данное средство и спаренная с ним утилита

patch(1) определили стиль распределенной сетевой разработки современной операционной системы Unix. Очень ценным свойством программы diff является то, что она нечасто удивляет пользователей. В ней отсутствуют частные случаи или сложные граничные условия, поскольку используется простой, математически совершенный метод сравнения последовательностей. Из этого можно сделать ряд выводов.

Благодаря математической модели и цельному алгоритму, Unix-утилита diff заметно контрастирует со своими подражателями. Во-первых, центральное ядро является цельным, небольшим и никогда не требовало ни единой строки для обслуживания. Во-вторых, результаты ее работы являются четкими и последовательными, не искажены сюрпризами, при которых эвристические методы терпят неудачу. Дуг Макилрой.

Таким образом, у пользователей программы diff может развиться интуитивное чувство относительно того, что будет делать программа в любой ситуации, даже без полного понимания центрального алгоритма. В Unix имеется множество других широко известных примеров, подтверждающих это. Ниже приводятся некоторые из них.

• Утилита

grep(1) для выбора из файлов строк, соответствующих шаблону, является простым упаковщиком вокруг формальной алгебры шаблонов регулярных выражений (описание приведено в разделе 8.2.2). Если бы данная программа испытывала недостаток в такой последовательной математической модели, то она, вероятно, выглядела бы подобно конструкции первоначального средства старейших Unix-систем,

glob(1) — набор узкоспециальных шаблонов, которые невозможно было комбинировать.

•

yacc(1) — утилита для создания языковых анализаторов представляет собой тонкий упаковщик вокруг формальной теории грамматики LR(1). Сопутствующая ей утилита, генератор лексических анализаторов

lex(1) является подобным тонким упаковщиком вокруг теории недетерминированных конечных автоматов.

Все три описанные программы являются настолько "свободными от ошибок", что их корректная работа воспринимается как должное, и в то же время они считаются достаточно компактными, для того чтобы программисты могли их использовать. В основном именно благодаря тому, что данные программы были сконструированы вокруг устойчивой и обоснованно корректной алгоритмической основы, они никогда не нуждались в серьезной доводке в процессе длительного и частого использования.

Противоположный формальному подход заключается в использовании

эвристики — эмпирических правил, которые позволяют получить вероятностное, а не абсолютно точное решение. В некоторых случаях эвристика применяется ввиду того, что детерминированное корректное решение невозможно. В качестве примера можно рассмотреть методы фильтрации спама. Для алгоритмически идеального спам-фильтра потребовалось бы в качестве модуля полное решение проблемы понимания естественного языка. В других случаях эвристика используется из-за того, что известные формально корректные методы невероятно дороги. Примером этому служит управление виртуальной памятью. Существуют почти совершенные решения, однако они требуют такого количества измерений во время выполнения, что их издержки свели бы к нулю любой теоретический выигрыш по сравнению с эвристикой.

Проблема эвристических методов заключается в том, что они увеличивают количество частных и граничных случаев. Если ничего другого не остается, то обычно следует ограничить эвристику с помощью какого-либо механизма восстановления на случай сбоя. Все обычные проблемы, связанные с увеличением сложности, сохраняются. Для управления итоговым выбором оптимальных соотношений необходимо начать их изучение. Всегда следует выяснять, действительно ли эвристические методы дают такой выигрыш производительности, который окупает затраты на них, связанные со сложностью кода. При этом не стоит оценивать наугад прирост производительности.

4.2.5. Значение освобождения

В начале данной книги упоминалась ссылка на Дзэн об "особой передаче знаний вне священного писания". Не следует рассматривать ее как экзотическую, использованную ради стилистического эффекта. Основные понятия Unix всегда отличались свободной, Дзэн-подобной простотой. Данное качество отражено в основополагающих документах Unix, таких как книга

"The C Programming Language" [42], а также доклад CACM 1974 года, в котором Unix была "представлена миру". Приведем одну известную цитату из данного документа: "... ограничение побуждает не только к экономии, но и к определенному изяществу дизайна". Источником этой простоты было стремление думать не о том, как много язык программирования или операционная система способны сделать, а о том, как

мало они могут сделать.

Для того чтобы проектировать с учетом компактности и ортогональности, следует начинать с нуля. Дзэн учит, что привязанность приводит к мукам; опыт проектирования программного обеспечения учит, что привязанность к поверхностным предположениям ведет к неортогональным, некомпактным конструкциям и проектам, которые терпят неудачу или становятся крайне сложными при сопровождении.

Для достижения просветления и прекращения страданий Дзэн учит освобождаться. Традиция Unix преподает ценность освобождения от частных, случайных условий, с которыми была сформулирована проектная задача. Абстрагируйтесь. Упрощайте. Обобщайте. Поскольку программы создаются для разрешения проблем, полностью отделиться от проблем невозможно. Однако стоит приложить умственные усилия, для того чтобы понять, сколько предубеждений можно отбросить, и выяснить, становится ли конструкция более компактной и ортогональной в ходе этого процесса. Часто в результате появляются возможности для повторного использования кода.

Различные ассоциации между Unix и Дзэн[44] также являются частью традиции Unix, и это не случайно.

4.3. Иерархичность программного обеспечения

В проектировании иерархии функций или объектов определяется два направления. От выбора направления очень зависит иерархическое представление кода.

4.3.1. Сравнение нисходящего и восходящего программирования

Первое направление является восходящим от конкретной проблемы к абстракции и представляет собой разработку исходя из специфических операций в предметной области, которые необходимо реализовать. Например, если разрабатывается программно-аппаратное обеспечение для дискового накопителя, то некоторыми низкоуровневыми примитивами могут быть "подвод головки к физическому блоку", "чтение физического блока", "запись в физический блок", "переключение светодиодного индикатора диска" и другие.

Другое направление является нисходящим, от абстракции к конкретным функциям, от

высокоуровневых спецификаций, описывающих проект в целом или логику приложения, вниз к отдельным операциям. Следовательно, при проектировании программного обеспечения для контроллера запоминающего устройства большой емкости, который может оперировать несколькими различными типами носителей, можно начинать с определения абстрактных операций, таких как "поиск логического блока", "чтение логического блока", "запись логического блока", "переключение индикатора активности". Данные операции отличались бы от операций аппаратного уровня с подобными названиями тем, что они были бы предназначены в качестве общих для различных видов физических устройств.

Два описанных выше примера могут представлять два конструкторских подхода для одного и того же семейства аппаратного обеспечения. Выбор решения в подобных ситуациях представляется таким: абстрагировать аппаратное обеспечение (так чтобы объекты инкапсулировали реальные элементы внешнего мира, а программа просто представляла собой список операций по их обработке), либо организовать программу на основе некоторой поведенческой модели (а затем внедрить действительные аппаратные манипуляции, которые осуществляют эту модель в потоке поведенческой логики).

Аналогичный выбор обнаруживается во многих других ситуациях. Предположим, что разрабатывается программа цифрового музыкального синтезатора (MIDI sequencer). Организовывать такой код можно вокруг его верхнего уровня (упорядочение дорожек) или вокруг нижнего уровня (переключение фрагментов или выборок и управление генераторами колебаний).

Существует весьма точный способ анализа данных различий. Он заключается в том, чтобы выяснить, организована ли конструкция вокруг ее главного событийного цикла (стремящегося к тесной связи с высокоуровневой логикой приложения) или вокруг служебной библиотеки всех операций, которые может вызывать главный цикл. Проектировщик, разрабатывающий программу сверху вниз, начнет с обдумывания ее основного событийного цикла, а специфические события внедрит позднее. Проектировщик, работающий снизу вверх, начнет с обдумывания инкапсуляции специфических задач, а позднее соединит их в некую логическую последовательность.

В качестве более крупного примера рассмотрим разработку Web-браузера. Верхним уровнем конструкции Web-браузера является спецификация ожидаемого поведения данной программы: какие типы URL-адресов (http: или ftp:, или file:) она интерпретирует, какие виды изображений она способна визуализировать, допускает ли она использование языков Java или JavaScript и с какими ограничениями и т.д. Уровень реализации, который соответствует данному верхнему уровню, является основным событийным циклом программы. В каждой итерации цикл ожидает, накапливает и координирует действия пользователя (такие как нажатие Web-ссылки или ввод символа в поле).

Однако Web-браузер для решения поставленных перед ним задач вынужден вызывать большой набор основных примитивов. Одна группа примитивов занята установкой сетевых соединений, отправкой данных по ним, а также получением ответов. В другую группу входят операции GUI-инструментария, который используется браузером. Третья группа может быть занята механическим преобразованием полученных HTML-документов из текстовой формы в объектное дерево документа.

Важно то, с какой стороны этого набора начинается разработка, поскольку уровень на противоположной стороне, весьма вероятно, будет ограничен первоначальным выбором. В частности, если программа разрабатывается исключительно сверху вниз, то программист может оказаться в неудобном положении, когда примитивы предметной области, которые необходимы логике приложения, не совпадают с теми, которые фактически можно реализовать. С другой стороны, если программа разрабатывается исключительно снизу вверх, то может оказаться, что приходится проделывать большой объем работы, не

соответствующей логике приложения, или просто проектируется "штабель кирпичей", в то время как необходимо "построить здание".

С момента возникновения полемики по поводу структурного программирования в 60-х годах, начинающих программистов, как правило, учат, что верный подход заключается в проектировании сверху вниз. То есть в поэтапном усовершенствовании, где на абстрактном уровне определяется, для чего предназначена данная программа, и постепенном заполнении пустот реализации до тех пор, пока не будет образован точно работающий код. Нисходящий подход становится хорошей практикой, когда выполняются три предусловия: (a) можно с большой точностью определить, для решения каких задач предназначена данная программа, (b) значительные изменения спецификации в ходе реализации маловероятны и (c) большая свобода выбора на низком уровне, каким образом программа будет выполнять свои функции.

Данные условия наиболее часто выполняются в программах, расположенных сравнительно близко к пользователю и высоко в стеке программ, т.е. в прикладном программировании. Однако даже в этом случае указанные предусловия часто не выполняются. Невозможно заранее определить "правильный" режим работы текстового редактора или графической программы до тех пор, пока пользовательский интерфейс не пройдет тестирование среди конечных пользователей. Исключительно нисходящее программирование часто характеризуется чрезмерным вложением трудозатрат в код, который придется удалить за ненадобностью и перестроить, поскольку интерфейс не проходит проверку в реальных условиях.

Для того чтобы обезопасить себя, программисты пытаются использовать оба подхода — описывают абстрактную спецификацию как нисходящую логику приложения

и собирают множество низкоуровневых примитивов предметной области в функциях или библиотеках с тем, чтобы в случае изменения высокоуровневой конструкции их можно было использовать повторно.

Unix-программисты наследуют традицию, которая является центральной в системном программировании, где низкоуровневыми примитивами являются операции аппаратного уровня, которые имеют постоянный характер и чрезвычайно важны. Следовательно, "благодаря приобретенному инстинкту" они более склонны к восходящему программированию.

Независимо от того является программист системным или нет, восходящий подход также может выглядеть более привлекательно при программировании исследовательским способом, когда пытаются получить контроль над феноменами аппаратного или программного обеспечения или реальными феноменами, которые еще не полностью понятны. Восходящее программирование предоставляет время и пространство для уточнения нечеткой спецификации. Кроме того, восходящее программирование "апеллирует к естественной человеческой лени" программистов: когда требуется удалить часть кода и перестроить его, при работе в нисходящем направлении приходится удалять более крупные фрагменты, чем при работе в восходящем направлении.

Таким образом, создание реального кода склоняется к использованию как нисходящего, так и восходящего подходов. Нередко нисходящий и восходящий код является частью одного и того же проекта. В таком случае возникает необходимость использования связующих уровней.

4.3.2. Связующие уровни

Довольно часто столкновение нисходящего и восходящего подходов является причиной некоторого беспорядка. Верхний уровень логики приложения и нижний уровень основных примитивов необходимо согласовать с помощью уровня связующей логики.

Один из уроков, которые Unix-программисты осваивали десятилетиями, состоит в том, что связующая технология представляет собой опасное нагромождение, и жизненно важным является сохранение связующих уровней как можно более тонкими. Связующий уровень должен соединять другие уровни, но его не следует использовать для сокрытия "изломов" и "шероховатостей" в них.

В примере с Web-браузером связующий уровень включал бы в себя код визуализации, который преобразовывает объект документа, полученный из входящего HTML- файла в сглаженное визуальное изображение в виде растра в буфере экрана, используя для формирования изображения основные примитивы GUI-интерфейса. Данный код визуализации печально известен как наиболее подверженная ошибкам часть браузера. Он содержит в себе ухищрения, направленные на разрешение проблем, которые связаны как с синтаксическим анализом HTML-кода (ввиду большого количества неверно сформированной там разметки), так и в инструментальном наборе GUI (в котором могут отсутствовать действительно необходимые примитивы).

Связующий уровень Web-браузера должен служить промежуточным звеном не только между спецификацией и основными примитивами, но и между несколькими различными внешними спецификациями: работой сети, описанной в протоколе HTTP, структурой HTML-документа и различными графическими мультимедийными форматами, а также поведенческими ожиданиями пользователей при работе с GUI-интерфейсом.

Однако один единственный чреватый ошибками связующий уровень не является наибольшей проблемой. Разработчик, который знает о существовании связующего уровня и пытается организовать его в средний уровень вокруг собственного набора структур данных или объектов, может в итоге получить

два связующих уровня — один выше среднего уровня, а другой ниже. Талантливые, но неопытные программисты особенно склонны попадать в эту ловушку. Они правильно выбирают все основные наборы классов (логику приложения, средний уровень и примитивы предметной области) и переделывают их подобно примерам из учебников, и только сбиваются с пути по мере того, как величина нескольких связующих уровней, необходимых для интеграции всего привлекательного кода, становится все больше и больше.

Принцип тонкого связующего уровня можно рассматривать как уточнение правила разделения. Политика (т.е. логика приложения) должна быть четко обособлена от механизма (то есть основных примитивов), однако очень большой код, который не является ни политикой, ни механизмом, скорее всего, свидетельствует о том, что его функции минимальны и что он только увеличивает общую сложность в системе.

4.3.3. Учебный пример: язык C считается тонким связующим уровнем

Язык C является хорошим примером эффективности тонкого связующего уровня.

В конце 90-х годов Джеррит Блаау (Gerrit Blaauw) и Фред Брукс (Fred Brooks) в книге

"Computer Architecture: Concepts and Evolution" [4] отмечали, что архитектура всех поколений компьютеров (от ранних мэйнфреймов, мини-компьютеров и рабочих станций до PC)

стремилась к конвергенции. Чем более поздней была конструкция в своем технологическом поколении, тем плотнее она приближался к тому, что Блаау и Брукс назвали "классической архитектурой" (classical architecture): двоичное представление, линейное адресное пространство, разграничение памяти и рабочего хранилища (регистров), универсальные регистры, определение адреса, занимающего фиксированное число байтов, двухадресные команды, порядок следования байтов[45] и типы данных как последовательное множество с размерами, кратными 4, либо 6 (6-битовые семейства в настоящее время устарело).

Томпсон и Ритчи разрабатывали язык C как подобие структурированного ассемблера для идеализированной архитектуры процессора и памяти, которую, как они ожидали, можно было смоделировать на большинстве традиционных компьютеров. По счастливой случайности, их моделью для идеализированного процессора был компьютер PDP-11, весьма продуманная и изящная конструкция мини-компьютера, которая вплотную приближалась к классической архитектуре Блаау и Брукса. Здраво рассуждая, Томпсон и Ритчи отказались внедрять в C большинство характерных особенностей (таких как порядок байтов) там, где PDP-11 ему не соответствовал[46].

PDP-11 стал важной моделью для последующих поколений микропроцессоров. Оказалось, что базовые абстракции C весьма четко охватывают классическую архитектуру. Таким образом, C начинался как хорошее дополнение для микропроцессоров и, вместо того чтобы стать непригодным в связи с тем, что его предположения устарели, фактически становился

лучше, по мере того, как аппаратное обеспечение все более сильно сливалось с классической архитектурой. Одним примечательным примером этой конвергенции была замена в 1985 году процессора Intel 286 с неуклюжей сегментной адресацией памяти процессором серии 386 с большим простым адресным пространством памяти. Чистый язык C был действительно лучшим дополнением для процессоров 386, чем для процессоров 286-й серии.

Не случайно, что экспериментальная эра в компьютерной архитектуре завершилась в середине 80-х годов прошлого века, т.е. в то время, когда язык C (и его ближайший потомок C++) побеждали все предшествующие им универсальные языки программирования. Язык C, разработанный как тонкий, но гибкий уровень над классической архитектурой, выглядит в перспективе двух десятилетий как почти наилучшая из возможных конструкций для ниши структурированного ассемблера, которую он и должен был занять. В дополнение к компактности, ортогональности и независимости (от машинной архитектуры, на которой он первоначально был разработан), данный язык также имеет важное качество прозрачности, рассмотренное в главе 6. В конструкции нескольких языков программирования, которые, возможно, являются лучшими, потребовалось внести серьезные изменения (такие как введение функции сборки мусора в памяти), чтобы создать достаточную функциональную дистанцию от C и избежать вытеснения им.

Эту историю стоит вспоминать и переосмысливать, поскольку пример языка C показывает, насколько мощной может быть четкая, минималистская конструкция. Если бы Томпсон и Ритчи были менее дальновидными, то они создали бы язык, который делал бы гораздо больше, опирался бы на более строгие предположения, никогда удовлетворительно не переносился бы с исходной аппаратной платформы и исчез бы вместе с ней. Напротив, язык C расцвел, и с тех пор пример Томпсона и Ритчи влияет на стиль Unix-разработки. Однажды в беседе о конструировании самолетов, писатель, искатель приключений, художник и авиаинженер Антуан де Сент-Экзюпери подчеркнул: "Совершенство достигается не в тот момент, когда более нечего добавить, а тогда, когда нечего более удалить".

Ритчи и Томпсон жили по этому принципу. Долгое время после того как ресурсные ограничения на ранних Unix-программах были смягчены, они работали над тем, чтобы поддерживать C в виде настолько тонкого уровня над аппаратным обеспечением, насколько

это возможно.

Когда я просил о какой-либо особенно экстравагантной функции в C, Деннис обычно говорил мне: "Если тебе нужен PL/1, ты знаешь, где его взять". Ему не приходилось общаться с каким-либо маркетологом, утверждающим: "На диаграмме продаж нам нужна галочка в рамочке!". Майк Леск.

История C также подтверждает важность существования работающей эталонной реализации

до стандартизации. Повторно данная тема затрагивается в главе 17, где рассматривается развитие стандартов C и Unix.

4.4. Библиотеки

Одним из последствий того влияния, которое стиль Unix-программирования оказал на модульность и четко определенные API-интерфейсы, является устойчивая тенденция к разложению программ на фрагменты связующего уровня, объединяющего семейства библиотек, особенно общих библиотек (эквивалентов структур, которые в Windows и других операционных системах называются динамически подключаемыми библиотеками или DLL (Dynamically-Linked Libraries)).

Если подходить к проектированию тщательно и обдуманно, то часто возникает возможность разделить программу таким образом, чтобы она состояла из главной части поддержки пользовательского интерфейса (т.е. политики) и совокупности служебных подпрограмм (т.е. механизма) без связующего уровня вообще. Данный подход представляется особенно целесообразным в ситуации, когда программа должна выполнять большое количество узкоспециальных операций с такими структурами данных, как графические изображения, пакеты сетевых протоколов или блоки управления аппаратного интерфейса. В статье

"The Discipline and Method Architecture for Reusable Libraries" [87] собрано несколько общих полезных, конструктивных советов, исходящих из традиций Unix, особенно полезных для решения проблем управления ресурсами в библиотеках такого вида.

Практика, при которой подобное разделение на уровни осуществляется явно, в Unix-программировании является стандартной. При этом служебные подпрограммы собираются в библиотеку, которая документируется отдельно. В таких программах клиентская часть специализируется на задачах пользовательского интерфейса и протоколе высокого уровня. Несколько большего внимания к конструкции требует отделение оригинальной клиентской части и ее замена другими, адаптированными для иных целей. Некоторые другие преимущества позволит раскрыть учебный пример.

Существует оборотная сторона данной проблемы. В мире Unix библиотеки, поставляемые

как библиотеки , должны сопровождаться тестовыми программами.

API-интерфейсы должны сопутствовать программам и наоборот. API, для использования которого необходимо написать C-код и который невозможно без труда вызвать из командной строки, очень тяжело изучать и использовать. И наоборот, невероятно сложно использовать интерфейсы,

единственной открытой и документированной формой которых является какая-либо программа и которые невозможно просто вызвать из программы на C, — например,

route(1) в прежних Linux-системах.Генри Спенсер.

Кроме упрощения процесса обучения, тестовые программы библиотек часто создают превосходные тестовые структуры. Поэтому опытные Unix-программисты видят в них не только форму для приложения умственных усилий пользователя библиотеки, но и свидетельство того, что код, вероятно, был хорошо протестирован.

Важной формой создания иерархии библиотек является

подключаемая подпрограмма (plugin) — библиотека с набором известных входных точек, которая динамически загружается после запуска и предназначена для решения специализированной задачи. Для работы таких подпрограмм необходимо, чтобы вызывающая программа была организована в значительной степени как документированная служебная библиотека, в которую подключаемая подпрограмма может направить обратный вызов.

4.4.1. Учебный пример: подключаемые подпрограммы GIMP

Программа GIMP (GNU Image Manipulation program— программа обработки изображений) разрабатывалась как графический редактор с управлением посредством интерактивного GUI-интерфейса. Однако GIMP построена как библиотека подпрограмм обработки изображений и вспомогательных подпрограмм, которые вызываются сравнительно тонким уровнем управляющего кода. Управляющий код "знает" о GUI, но не имеет непосредственной информации о форматах изображений. Библиотечные подпрограммы, напротив, распознают форматы изображений, но не имеют информации о GUI-интерфейсе.

Библиотечный уровень документирован (и фактически распространяется в виде пакета "libgimp" для использования другими программами). Это означает, что программы на языке C, которые называются "подключаемыми подпрограммами", могут динамически загружаться в GIMP и вызывать библиотеку для обработки изображений, фактически принимая на себя управление на том же уровне, что и GUI-интерфейс (см. рис. 4.2).

Рис. 4.2. Связи между вызывающей и вызываемой программой в редакторе GIMP с загруженной подключаемой подпрограммой

Подключаемые подпрограммы используются для осуществления множества специализированных преобразований. В число таких преобразований входят: обработка карты цветов, размывание границ и очистка изображения; чтение и запись форматов, не характерных для ядра GIMP; такие расширения, как редактирование мультипликации и тем оконных менеджеров; многие другие виды обработки изображений, которые могут быть автоматизированы с помощью сценариев, содержащих логику обработки изображений в ядре GIMP. Перечень подключаемых подпрограмм для GIMP доступен в World Wide Web.

Несмотря на то, что большинство подключаемых подпрограмм для GIMP являются небольшими и простыми C-программами, существует также возможность написать подпрограмму, которая открывает библиотечный API-интерфейс для языков сценариев. Данная возможность описывается в главе 11 в ходе изучения модели "многопараметрических (polyvalent) программ".

С середины 80-х годов прошлого века большинство новых конструкций языков обладают собственной поддержкой

объектно-ориентированного программирования (Object-Oriented Programming — OO) . Напомним, что в объектно-ориентированном программировании функции, воздействующие на определенную структуру данных, инкапсулируются вместе с данными в объект, который может рассматриваться как единое целое. В противоположность этому, модули в не-OO-языках делают связь между данными и воздействующими на них функциями весьма второстепенной, и модули часто воздействуют на данные и внутреннее устройство других модулей.

Первоначально идея OO-дизайна позитивно сказалась в конструировании графических систем, графических пользовательских интерфейсов и определенных видов моделирования. К удивлению и постепенному разочарованию многих, обнаружились трудности в проявлении существенных преимуществ вне этих областей. Причины этого заслуживают отдельного рассмотрения.

Существует некоторый конфликт между Unix-традицией модульности и моделями использования, которые развились вокруг OO-языков. Unix-программисты всегда несколько более скептически относились к OO-технологии, чем их коллеги, работающие в других операционных системах. Частично из-за правила разнообразия. Слишком часто OO-подход объявлялся единственно верным решением проблемы сложности программного обеспечения. Однако здесь кроется еще одна проблема, которую стоит исследовать, прежде чем оценивать определенные OO (объектно-ориентированные) языки в главе 14. Рассмотрение этой проблемы также поможет лучше описать некоторые характеристики Unix-стиля не-OO-программирования.

Выше отмечалось, что Unix-традиция модульности является традицией тонкого связующего, минималистского подхода с несколькими уровнями абстракции между аппаратным обеспечением и объектами верхнего уровня программ. Частично это является влиянием языка C. Моделирование истинных объектов в языке C обычно сопряжено с большими усилиями. Вследствие этого нагромождение уровней абстракции является утомительным. Поэтому иерархии объектов в C склонны к относительной простоте и прозрачности. Даже применяя другие языки, Unix-программисты склонны переносить стиль использования тонкого связующего уровня и простой иерархии, которому они научились, используя Unix-модели.

OO-языки упрощают абстракцию, возможно, даже слишком упрощают. Они поддерживают создание структур с большим количеством связующего кода и сложными уровнями. Это может оказаться полезным в случае, если предметная область является действительно сложной и требует множества абстракций, и вместе с тем такой подход может обернуться неприятностями, если программисты реализуют простые вещи сложными способами, просто потому что им известны эти способы и они умеют ими пользоваться.

Все OO-языки несколько склонны "втягивать" программистов в ловушку избыточной иерархии. Объектные структуры и браузеры объектов не являются заменой хорошего дизайна или документации, но часто рассматриваются как таковые. Чрезмерное количество уровней разрушает прозрачность: крайне затрудняется их просмотр и анализ ментальной модели, которую по существу реализует код. Всецело нарушаются правила простоты, ясности и прозрачности, а в результате код наполняется скрытыми ошибками и создает постоянные проблемы при сопровождении.

Данная тенденция, вероятно, усугубляется тем, что множество курсов по программированию

преподают громоздкую иерархию как способ удовлетворения правила представления. С этой точки зрения множество классов приравниваются к внедрению знаний в данные. Проблема данного подхода заключается в том, что слишком часто "развитые данные" в связующих уровнях фактически не относятся к какому-либо естественному объекту в области действия программы — они предназначены только для связующего уровня. (Одним из верных признаков этого является распространение абстрактных подклассов или "смесей".)

Другим побочным эффектом ОО-абстракции представляется то, что постепенно исчезают возможности для оптимизации. Например,

a+a+a+a может стать

a*4 или даже

a<<2 , в случае если a — целое число. Однако если кто-либо создаст класс с операторами, то ничто не будет указывать на то, являются ли они коммутативными, дистрибутивными или ассоциативными. Так как создатель класса не предполагал показывать внутреннее устройство объекта, то невозможно определить, какое из двух эквивалентных выражений более эффективно. Само по себе это не является достаточной причиной избегать использования ОО-методик в новых проектах; это было бы преждевременной оптимизацией. Однако это причина подумать дважды, прежде чем преобразовывать не-ОО-код в иерархию классов.

Для Unix-программистов характерно инстинктивное осознание данных проблем. Данная тенденция представляется одной из причин, по которой ОО-языкам в Unix не удалось вытеснить не-ОО-конструкции, такие как C, Perl (который в действительности обладает ОО-средствами, но они используются не широко) и shell. В мире Unix больше открытой критики ОО-языков, чем это позволяют ортодоксы в других операционных системах. Unix-программисты знают, когда

не использовать объектно-ориентированный подход, а, если они действительно используют ОО-языки, то тратят большие усилия, пытаясь сохранить объектные конструкции четкими. Как однажды (в несколько другом контексте) заметил автор книги

"The Elements of Networking Style" [60]: "... если программист знает, что делает, то трех уровней будет достаточно, если же нет, то не помогут даже семнадцать уровней".

Одной из причин того, что ОО-языки преуспели в большинстве характерных для них предметных областей (GUI-интерфейсы, моделирование, графические средства), возможно, является то, что в этих областях относительно трудно неправильно определить онтологию типов. Например, в GUI-интерфейсах и графических средствах присутствует довольно естественное соответствие между манипулируемыми визуальными объектами и классами. Если выясняется, что создается большое количество классов, которые не имеют очевидного соответствия с тем, что происходит на экране, то, соответственно, легко заметить, что связующий уровень стал слишком большим.

Одна из основных трудностей проектирования в стиле Unix состоит в комбинации достоинств отделения (упрощение и обобщение проблем из их исходного контекста) с достоинствами тонкого связующего уровня и плоских, простых и прозрачных иерархий кода и конструкции.

Некоторые из этих моментов будут повторно рассматриваться при обсуждении объектно-ориентированных языков программирования в главе 14.

4.6. Создание модульного кода

Модульность выражается в хорошем коде, но главным образом она является следствием хорошего проектирования. Ниже приведен ряд вопросов о разрабатываемом коде, ответы на которые могут помочь программисту в улучшении модульности кода.

• Сколько глобальных переменных присутствует в коде? Глобальные переменные — разрушители модульности, простой способ передачи информации из одних компонентов в другие неаккуратным и беспорядочным путем[47].

• Остаются ли размеры отдельных модулей в пределах зоны наилучшего восприятия Хаттона? Если это не так, то возможно появление долговременных проблем при сопровождении. Известны ли пределы зоны наилучшего восприятия для данного программиста? Известны ли пределы зоны для других сотрудничающих программистов? Если нет, то наилучшим решением будет придерживаться консервативной точки зрения и сохранять размеры, ближайшие к нижней границе диапазона Хаттона.

• Не слишком ли крупные отдельные функции в модулях? Это не столько вопрос количества строк кода, сколько его внутренней сложности. Если неформально в одной строке невозможно описать взаимодействие функции и вызывающей ее программы, то, вероятно, размер функции слишком велик[48].

Лично я склонен разбивать подпрограмму, когда в ней слишком много локальных переменных. Другой признак — уровни отступов (их слишком много). Я редко смотрю на длину. Кен Томпсон.

• Имеет ли код внутренние API-интерфейсы — т.е. наборы вызовов функций и структур данных, которые можно описать как цельные блоки, каждый из которых изолирует некоторый уровень функций от остальной части кода? Хороший API имеет смысл и понятен без рассмотрения скрытой в нем реализации. Классическая проверка заключается в том, чтобы попытаться описать API другому программисту по телефону. Если это не удалось, то весьма вероятно, что интерфейс слишком сложен и спроектирован неудачно.

• Имеет ли любой из разрабатываемых API-интерфейсов более семи входных точек? Имеет ли какой-либо из классов более семи методов? Имеют ли структуры данных более семи членов?

• Каково распределение входных точек в каждом модуле проекта?[49] Не кажется ли распределение неравномерным? Действительно ли в некоторых модулях необходимо такое большое количество входных точек? Сложность модуля также растет, как квадрат числа входных точек — еще одна причина того, что простые API лучше, чем сложные.

Может оказаться полезным сравнение данных вопросов с перечнем вопросов о прозрачности и воспринимаемости в главе 6.

5

Текстовое представление данных: ясные протоколы лежат в основе хорошей практики

В данной главе рассматриваются традиции Unix в аспекте двух различных, но тесно связанных друг с другом видов проектирования: проектирования форматов файлов для сохранения данных приложений в постоянном хранилище памяти и проектирования протоколов прикладного уровня для передачи (возможно, через сеть) данных и команд между

взаимодействующими программами.

Объединяет оба вида проектирования то, что они задействуют сериализацию структур данных. Для внутренней работы компьютерных программ наиболее удобным представлением сложной структуры данных является то, в котором все поля имеют характерный для конкретной машины формат данных (например, представление целых чисел со знаком в двоичном дополнительном коде) и все указатели являются абсолютными адресами памяти (в противоположность, например, именованным ссылкам). Однако такие формы представления не подходят для хранения и передачи. Адреса памяти в структуре данных теряют свое значение за пределами оперативной памяти, и выпуск необработанных собственных форматов данных приводит к проблемам взаимодействия при передаче данных между машинами с различными соглашениями (например, с обратным и прямым порядком следования байтов или между 32- и 64-битовой архитектурами).

Для передачи и хранения передаваемое квази-пространственное расположение структур данных, таких как связные списки, должно быть сглажено или сериализовано в представление потока байтов, из которого впоследствии можно будет восстановить исходную структуру. Операция сериализации (сохранения) иногда называется

маршалингом (marshaling) , а обратная ей операция (загрузка) —

демаршалингом (unmarshaling) . Данные термины применимы по отношению к объектам в ОО-языках программирования, таких как C++, Python или Java, вместе с тем они в равной степени применимы для таких операций, как загрузка графического файла во внутреннюю память графического редактора и сохранение файла после модификации.

Значительной частью того, что поддерживают программисты на C и C++, является специальный код для операций маршалинга и демаршалинга, даже если выбранная форма представления для сохранения и восстановления также проста как дамп бинарной структуры (распространенная методика в не Unix-средах). Современные языки, такие как Python и Java, имеют встроенные функции демаршалинга и маршалинга, которые применимы к любому объекту или потоку байтов, представляющему объект, и в значительной степени сокращают трудозатраты.

Однако эти простые методы часто неудовлетворительны в силу различных причин, включая упомянутые выше проблемы взаимодействия между машинами, а также ту негативную особенность, которая связана с их непрозрачностью для других средств. В ситуации, когда приложение представляет собой сетевой протокол, исходя из соображений экономичности, иногда целесообразно представлять внутреннюю структуру данных (такую, например, как сообщение с адресами отправителя и получателя) не в виде одного большого двоичного объекта данных, а в виде последовательности транзакций или сообщений, которые могут быть отклонены принимающей машиной (так что, например, большое сообщение, может быть отклонено, если адрес получателя указан неверно).

Способность к взаимодействию, прозрачность, расширяемость и экономичность хранения или транзакций — важнейшие темы в проектировании форматов файлов и прикладных протоколов. Способность к взаимодействию и прозрачность требуют, чтобы при проектировании таких конструкций основное внимание было уделено четким формам представления данных, а не удобству реализации или максимальной производительности. Расширяемость также благоприятствует текстовым протоколам, так как двоичные протоколы часто труднее расширять или четко разделять на подмножества. Экономичность транзакций иногда заставляет двигаться в противоположном направлении, однако следует понимать, что выдвижение данного признака на первый план является некоторой формой преждевременной оптимизации. Более дальновидным будет все же противостоять ей.

Наконец, необходимо отметить отличие между форматами файлов данных и конфигурационных файлов, которые часто используются для установки параметров запуска Unix-программ. Самое основное отличие заключается в том, что (за редкими исключениями, такими как конфигурационный интерфейс редактора GNU Emacs) программы обычно не изменяют свои конфигурационные файлы — информационный поток является односторонним (от файла, считываемого при запуске, к настройкам приложения). С другой стороны, форматы файлов данных связывают свойства с именованными ресурсами, и такие файлы считываются и записываются соответствующими приложениями. Конфигурационные файлы, как правило, редактируются вручную и имеют небольшие размеры, тогда как файлы данных генерируются программами и могут достигать произвольных размеров.

Исторически Unix обладает связанными, но различными наборами соглашений для данных двух видов представления. Соглашения для конфигурационных файлов рассматриваются в главе 10; в данной главе описываются только соглашения для файлов данных.

5.1. Важность текстовой формы представления

Каналы и сокеты передают двоичные данные так же, как текст. Однако есть важные причины, для того чтобы примеры, рассматриваемые в главе 7, были текстовыми: причины, связанные с рекомендацией Дуга Макилроя, приведенной в главе 1. Текстовые потоки являются ценным универсальным форматом, поскольку они просты для чтения, записи и редактирования человеком без использования специализированных инструментов. Данные форматы прозрачны (или могут быть спроектированы как таковые).

Кроме того, сами ограничения текстовых потоков способствуют усилению инкапсуляции. Препятствуя сложным формам представления с богатой, плотно закодированной структурой, текстовые потоки также препятствуют созданию программ, которые смешивают внутренние компоненты друг друга. Текстовые потоки также способствуют усилению инкапсуляции. Эта проблема рассматривается в главе 7 при обсуждении технологии RPC.

Если программист испытывает острое желание разработать сложный двоичный формат файла или сложный двоичный прикладной протокол, как правило, мудрым решением будет приостановить работу до тех пор, пока это желание не пройдет. Если основной целью такой разработки является производительность, то внедрение сжатия текстового потока на каком-либо уровне выше или ниже данного протокола прикладного уровня предоставит аккуратную и, вероятно, более производительную конструкцию, чем двоичный протокол (текст хорошо и быстро сжимается).

Плохим примером двоичных форматов в истории Unix был способ аппаратно-независимого чтения программой

troff двоичного файла (предположительно в целях повышения скорости), содержащего информацию устройства. В первоначальной реализации данный двоичный файл генерировался из текстового описания способом, отчасти не пригодным для переноса на другие платформы. Столкнувшись с необходимостью

быстро перенести

ditroff на новую машину, я вместо того, чтобы переделывать двоичный материал, вырезал его и просто заставил

ditroff читать текстовый файл. Тщательно созданный код чтения файла сделал потерю

скорости незначительной.Генри Спенсер.

Проектирование текстового протокола часто защищает систему в будущем, поскольку диапазоны в числовых полях не подразумеваются самим форматом. В двоичных форматах обычно определяется количество битов, выделенных для данного значения, и расширение таких форматов является трудной задачей. Например, протокол IPv4 допускает использование только 32-битового адреса. Для того чтобы увеличить размер до 128 бит (как это сделано в протоколе IPv6), требуется значительная реконструкция[50]. Напротив, если требуется ввести большее значение в текстовый формат, то его необходимо просто записать. Возможно, что какая-либо программа не способна принимать значения в данном диапазоне, однако, программу обычно проще модифицировать, чем изменять все данные, хранящиеся в этом формате.

Если планируется манипулировать достаточно большими блоками данных, применение двоичного протокола можно считать оправданным, когда разработчик действительно заботится о наибольшей плотности записи имеющегося носителя, или когда существуют жесткие ограничения времени или инструкций, необходимых для интерпретации данных в структуру ядра. Форматы для больших изображений и мультимедиа данных в некоторых случаях являются примерами первого случая, а сетевые протоколы с жесткими ограничениями задержки — примером второго.

Аналогичная для SMTP- или HTTP-подобных текстовых протоколов проблема заключается в том, что они требуют большой полосы пропускания и их синтаксический анализ производится медленно. Наименьший X-запрос занимает 4 байта, а наименьший HTTP-запрос занимает около 100 байт. X-запросы, включая транспортные издержки, могут быть выполнены с помощью приблизительно 100 инструкций; разработчики Apache (Web-сервера) гордо заявляют, что сократили выполнение запроса до 7000 инструкций. Решающим фактором на выходе для графических приложений становится полоса пропускания. Аппаратное обеспечение разрабатывается таким образом, что в настоящее время шина графической платы является

бутылочным горлышком для небольших операций, поэтому любой протокол, если он не должен быть еще худшим бутылочным горлышком, должен быть очень компактным. Это предельный случай.Джим Геттис.

Данные вопросы справедливы и в других предельных случаях, в системе X Window, например, при проектировании форматов графических файлов, предназначенных для хранения очень больших изображений. Однако они обычно также свидетельствуют о преждевременной оптимизации. Текстовые форматы не обязательно имеют более низкую плотность записи, чем двоичные. В них все-таки используются семь из восьми битов каждого байта. И выигрыш в связи с тем, что не требуется осуществлять синтаксический анализ текста, как правило, нивелируется, когда впервые приходится создавать тестовую нагрузку или пристально изучать пример формата, сгенерированный программой.

Кроме того, проектирование компактных двоичных форматов значительно затруднено, когда необходимо сделать их четко расширяемыми. С данной проблемой столкнулись разработчики X Window.

Идее современного каркаса X противостоит тот факт, что мы не спроектировали достаточную структуру, для того чтобы упростить игнорирование случайных расширений протокола. Возможно когда-нибудь мы это сделаем, но было бы хорошо иметь несколько лучший каркас. Джим Геттис.

Когда разработчик полагает, что столкнулся с предельным случаем, оправдывающим двоичный формат файлов или протокол, следует предусмотреть и возможность расширения

пространства в конструкции, необходимого для дальнейшего роста.

5.1.1. Учебный пример: формат файлов паролей в Unix

Во многих операционных системах данные пользователей, необходимые для регистрации и запуска пользовательского сеанса, представляют собой трудную для понимания двоичную базу данных. В противоположность этому, в операционной системе Unix такие данные содержатся в текстовом файле с записями, каждая из которых является строкой, разделенной на поля с помощью знаков двоеточия.

В приведенном ниже примере содержатся некоторые случайно выбранные строки. Пример 5.1. Пример файла паролей

```
games:*:2:100:games:/usr/games:

gopher:*:13:30:gopher:/usr/lib/gopher-data:

ftp:*:14:50:FTP User:/home/ftp:

esr:0SmFuPnH5JINs:23:23:Eric S. Raymond:/home/esr:

nobody:*:99:99:Nobody:/:
```

Даже не зная ничего о семантике полей, можно отметить, что более плотно упаковать данные в двоичном формате было бы весьма трудно. Ограничивающие поля символы двоеточия должны были бы иметь функциональные эквиваленты, которые, как минимум, занимали бы столько же места (обычно либо определенное количество байтов, либо строки нулевой длины). Каждая запись, содержащая данные одного пользователя, должна была бы иметь ограничитель (который едва ли мог бы быть короче, чем один символ новой строки), либо неэкономно заполняться до фиксированной длины.

В действительности, перспективы сохранения пространства посредством двоичного кодирования почти полностью исчезают, если известна фактическая семантика данных. Значения числового идентификатора пользователя (третье поле) и идентификатора группы (четвертое поле) являются целыми числами, поэтому на большинстве машин двоичное представление данных идентификаторов заняло бы по крайней мере 4 байта и было бы длиннее текста для всех значений до 999. Это можно проигнорировать и предположить наилучший случай, при котором значения числовых полей находятся в диапазоне 0-255.

В таком случае можно было бы уплотнить числовые поля (третье и четвертое) путем сокращения каждого числа до одного байта и восьмибитового кодирования строки пароля (второе поле). В данном примере это дало бы около 8% сокращения размера.

8% мнимой неэффективности текстового формата имеют весьма большое значение. Они позволяют избежать наложения произвольного ограничения на диапазон числовых полей. Они дают возможность модифицировать файл паролей, используя любой старый предпочтительный текстовый редактор, т.е. освобождают от необходимости создавать специализированный инструмент для редактирования двоичного формата (хотя непосредственно в случае файла паролей необходимо быть особенно осторожным с одновременным редактированием). Кроме того, появляется возможность выполнять специальный поиск, фильтрацию и отчеты по учетной информации пользователей с помощью средств обработки текстовых потоков, таких как

grep(1) .

Действительно, необходимо быть осторожным, чтобы не вставить символ двоеточия в какое-либо текстовое поле. Хорошей практикой является создание такого кода записи файла, который предваряет вставляемые символы двоеточия знаком переключения (escape character), а код чтения файла затем интерпретирует данный символ. В традиции Unix для этих целей предпочтительно использовать символ обратной косой черты.

Тот факт, что структурная информация передается с помощью позиции поля, а не с помощью явной метки, ускоряет чтение и запись данного формата, но сам формат становится несколько жестким. Если ожидается изменение набора свойств, связанных с ключом, с любой частотой, то, вероятно, лучше подойдет один из теговых форматов, которые описываются ниже.

Экономичность не является главной проблемой файлов паролей, с которой следовало бы начинать обсуждение, поскольку такие файлы обычно считываются редко[51] и нечасто модифицируются. Способность к взаимодействию в данном случае не является проблемой, поскольку некоторые данные в файле (особенно номера пользователей и групп) не переносятся с машины, на которой они были созданы. Таким образом, в случае файлов паролей совершенно ясно, что следование критериям прозрачности было правильным.

5.1.2. Учебный пример: формат файлов .newsrc

Новости Usenet представляют собой распределенную по всему миру систему электронных досок объявлений, которая предвосхитила современные P2P-сети за два десятилетия до их появления. В Usenet используется формат сообщений, очень сходный с форматом сообщений электронной почты спецификации RFC 822, за исключением того, что вместо отправки непосредственно отдельным получателям, сообщения отправляются в тематические группы. Статьи, отправленные с одного из участвующих узлов, широковещательно распространяются каждому узлу, который зарегистрирован в качестве соседнего, и в конечном итоге достигают всех узлов группы новостей.

Почти все программы для чтения Usenet-новостей распознают файл .newsrc, в котором записывается, какие Usenet-сообщения просматривает вызывающих пользователь. Несмотря на то, что данный файл имеет имя, подобное файлу конфигурации, он не только считывается во время запуска, но, как правило, обновляется в конце сеанса программы. Формат .newsrc зафиксирован с момента появления первых программ чтения новостей, приблизительно в 1980 году. В примере 5.2. представлен характерный фрагмент файла .newsrc.

В каждой строке устанавливаются свойства для группы новостей, имя которой задается в первом поле. За именем следует специальный знак о подписке. Двоеточие указывает на ее наличие, а восклицательный знак — на ее отсутствие. В остальной части строки содержится последовательность разделенных запятыми номеров или диапазонов номеров сообщений, указывающая на то, какие статьи были просмотрены пользователем.

Программисты, пишущие не для Unix, возможно, автоматически попытаются спроектировать быстрый двоичный формат, в котором состояние каждой группы новостей описано либо длинной двоичной записью фиксированной длины, либо последовательностью самоописательных двоичных пакетов с внутренними полями длины. Для того чтобы избежать издержек на синтаксический анализ всего диапазона выражений на этапе запуска, сутью

такого двоичного представления было бы выражение диапазонов с двоичными данными в спаренных полях длиной в одно слово. Пример 5.2. Файл .newsrc

rec.arts.sf.misc! 1-14774,14786,14789

rec.arts.sf.reviews! 1-2534

rec.arts.sf.written: 1-876513

news.answers! 1-199359,213516,215735

news.announce.newusers! 1-4399

news.newusers.questions! 1-645661

news.groups.questions! 1-32676

news.software.readers! 1-95504,137265,137274,140059,140091,140117

alt.test! 1-1441498

Запись и считывание файлов подобного формата могли бы осуществляться быстрее по сравнению с текстовыми файлами, но тогда возникали бы другие проблемы. Простая реализация в записях фиксированной длины создавала бы искусственные ограничения относительно длины имен групп новостей и (что более важно) на максимальное количество диапазонов номеров просматриваемых статей. Более сложный формат двоичных пакетов позволил бы избежать ограничений относительно длины, однако его невозможно было бы редактировать с помощью простых средств, а это очень важно, когда необходима переустановка только некоторых из битов чтения в отдельной группе новостей. Кроме того, данный формат не обязательно был бы переносимым на другие типы машин.

Разработчики первоначальной программы чтения новостей предпочли экономии прозрачность и способность к взаимодействию. Движение в другом направлении не было полностью ошибочным; файлы .newsrc могут достигать весьма больших размеров, и в одной из современных программ для чтения новостей (Pan в среде GNOME) используется оптимизированный по скорости частный формат, который позволяет избежать запаздывания при запуске. Но для других разработчиков в 1980 году текстовое представление было хорошим компромиссом и приобретало еще больший смысл по мере того, как скорость машин увеличивалась, а цены на накопительные устройства падали.

5.1.3. Учебный пример: PNG — формат графических файлов

PNG (Portable Network Graphics — переносимая сетевая графика) представляет собой формат для хранения растровых изображений. Он подобен GIF, и, в отличие от JPEG, в данном формате используется алгоритм сжатия без потерь. Формат PNG оптимизирован скорее для таких прикладных задач, как штриховая графика и пиктограммы, чем для фотографических изображений. Документация и высокого качества справочные библиотеки с открытым исходным кодом доступны на Web-сайте Portable Network Graphics <http://libpng.org/pub/png>.

PNG является превосходным примером вдумчиво спроектированного двоичного формата. Использование двоичного формата в данном случае целесообразно, поскольку графические файлы могут содержать такие большие объемы данных, при которых занимаемое

пространство и время Internet-загрузки значительно выросли бы, если бы информация о пикселях хранилась в текстовом виде. Первостепенная значимость придавалась экономичности транзакций за счет недостаточной прозрачности[52]. Однако разработчики позаботились о возможности взаимодействия. В PNG определяется порядок байтов, полная длина слова, порядок следования байтов и заполнение между полями (которое считается недостатком).

PNG-файл состоит из последовательности больших блоков данных, каждый из которых представлен в самоописательном формате и начинается с названия типа блока и длины блока. Благодаря такой организации нет необходимости включать в PNG-формат номер версии. Новые типы блоков могут быть добавлены в любое время. Регистр первой литеры в имени типа сообщает использующему PNG программному обеспечению о возможности безопасно игнорировать данный блок.

Заголовок PNG-файла также заслуживает изучения. Он продуманно спроектирован, для того чтобы упростить обнаружение различных распространенных видов повреждения файлов (например, в 7-битовых каналах передачи или при отсечении символов CR и LF).

Стандарт PNG можно определить как точный, завершенный и хорошо описанный. Он вполне мог бы послужить эталоном при написании стандартов файловых форматов.

5.2. Метаформаты файлов данных

Метаформат файлов данных представляет собой набор синтаксических и лексических соглашений, которые либо формально стандартизированы, либо достаточно хорошо "укоренились" в практике, и поэтому существуют стандартные служебные библиотеки для осуществления операций маршалинга и демаршалинга.

В операционной системе Unix развились или были заимствованы метаформаты, пригодные для широкого спектра прикладных задач. Хорошей практикой является использование одного из них (вместо какого-либо уникального частного формата) везде, где это возможно. Преимущества начинаются с количества частного кода для синтаксического анализа и создания файлов, написания которого можно избежать, используя служебную библиотеку. Однако наиболее важным преимуществом является то, что разработчики и даже многие пользователи немедленно распознают данные форматы и могут их удобно использовать, что сокращает издержки, связанные с изучением новых программ.

При последующем изложении ссылка на "традиционные инструментальные средства Unix" означает комбинацию утилит

grep(1), sed(1), awk(1), tr(1) и

cut(1) для выполнения поиска и преобразования текста. Perl и другие языки сценариев имеют собственную поддержку синтаксического анализа построчных форматов, поддерживаемых данными средствами.

Ниже представлены стандартные форматы, которые могут послужить в качестве моделей.

5.2.1. DSV-стиль

Аббревиатура DSV расшифровывается как

Delimiter-Separated Values (формат с разделителями значений) . В первом учебном примере рассматривался файл /etc/passwd, имеющий DSV-формат с символом двоеточия в качестве разделителя значений. В операционной системе Unix двоеточие является стандартным разделителем для DSV-форматов, в которых значения полей могут содержать пробелы.

Формат файла /etc/passwd (одна запись в строке, поля разделены двоеточиями) является весьма традиционным в Unix и часто используется для данных, представленных в виде таблиц. Другие классические примеры включают в себя файл /etc/group, описывающий группы пользователей, и файл /etc/inittab, который применяется для управления запуском и остановом служебных программ в Unix на различных уровнях выполнения операционной системы.

Ожидается, что организованные в таком стиле файлы данных поддерживают включение в поля данных символов двоеточия, предваренных символами обратной косой черты. В более общем смысле ожидается, что считывающий данные код поддерживает продолжение записи путем исключения знака переключения для символов начала новой строки и позволяет включать данные, содержащие непечатаемые символы, используя знаки переключения в стиле C.

Данный формат является наиболее подходящим в ситуациях, когда данные имеют табличную организацию, снабжены ключами (именами в первом поле), а записи, как правило, короткие (менее 80 символов). Описываемый формат хорошо обрабатывается с помощью традиционных инструментальных средств Unix.

Иногда встречаются и другие разделители полей, такие как символ канала (|) или даже символ ASCII NUL. В практике Unix старой школы привычно было поддерживать символы табуляции — форма представления, которая отражена в установках по умолчанию для утилит

cut(1) и

paste(1) . Однако постепенно данная форма представления изменялась, по мере того как разработчики форматов осознавали множество мелких неудобств, возникающих ввиду того, что символы табуляции и пробелы визуально неразличимы.

DSV-формат для Unix является тем же, чем CSV (формат с разделением значений запятыми) для Microsoft Windows и других систем вне мира Unix. Формат CSV (поля разделены запятыми, для буквального представления запятых используются двойные кавычки, продолжающиеся строки не поддерживаются) в Unix встречается нечасто.

В сущности, Microsoft-версия CSV представляет собой азбучный пример того, как

не следует проектировать текстовый файловый формат. Проблемы, связанные с ним, начинаются с ситуации, когда разделяющий символ (в данном случае запятая) находится внутри поля. В Unix в таком случае для буквального представления разделителя перед ним был бы вставлен символ обратной косой черты, а буквальная обратная косая черта представлялась бы при помощи двойной обратной косой черты. Такая конструкция создает единственный частный случай (знак переключения), который необходимо проверять во время синтаксического анализа файла, и требует единственного действия, когда такой символ найден, а именно — интерпретировать следующий символ буквально. Данное действие не только обрабатывает разделяющий символ, но и предоставляет способ обработки знака переключения и символов новой строки без дополнительных ухищрений. С другой стороны, в формате CSV целое поле заключается в двойные кавычки, в случае если оно содержит символ-разделитель. Если поле содержит двойные кавычки, его так же необходимо

заключать в двойные кавычки, а отдельные двойные кавычки в поле необходимо повторять дважды, для того чтобы указать, что они не завершают поле.

Существует два негативных результата роста числа частных случаев. Во-первых, возрастает сложность синтаксического анализатора (и его чувствительность к ошибкам). Во-вторых, ввиду того, что правила формата сложны и непредусмотрены, различные реализации расходятся в обработке граничных случаев. Иногда продолжающиеся строки

поддерживаются путем начала последнего поля строки с незакрытых двойных кавычек, но только в некоторых продуктах. Microsoft имеет несовместимые версии CSV-файлов между своими собственными приложениями, а в некоторых случаях между различными версиями одного приложения (очевидный пример — программа Excel).

5.2.2. Формат RFC 822

Метаформат RFC 822 происходит от текстового формата сообщений электронной почты в Internet. RFC 822 является основным Internet RFC-стандартом, описывающим данный формат (впоследствии заменен RFC 2822). Формат MIME (Multipurpose Internet Media Extension — многоцелевые расширения Internet) обеспечивает способ внедрения типизированных двоичных данных внутрь сообщений формата RFC 822. (Web-поиск по какому-либо из упомянутых названий предоставит ссылки на соответствующие стандарты).

В данном метаформате атрибуты записей хранятся по одному в строке, называются по меткам, имеющим сходство с именами полей в заголовке почтового сообщения, и ограничиваются символом двоеточия с последующим пробелом. Имена полей не содержат пробелов, традиционно вместо пробелов используется дефис. Значением атрибута является вся оставшаяся строка за исключением завершающего пробела и символа новой строки. Физическая строка, начинающаяся с символа табуляции или пробела, интерпретируется как продолжение текущей логической строки. Пустая строка может интерпретироваться либо как ограничитель записи, либо как указатель на то, что далее следует неструктурированный текст.

В операционной системе Unix метаформат RFC 822 является традиционным и предпочтительным для классифицированных сообщений или файлов, близко сопоставимых с электронной почтой. Более широко данный формат целесообразно использовать для записей с изменяющимся набором полей, в котором иерархия данных проста (без рекурсии или древовидной структуры).

Данный формат используется в группах новостей Usenet, как и в форматах HTTP 1.1. (и более поздних), используемых в World Wide Web. Он весьма удобен для редактирования вручную. Традиционные средства поиска в Unix хорошо проявляют себя в поиске атрибутов, хотя определение границ записей требует несколько больших усилий, чем это необходимо для построчного формата записей.

Недостатком формата RFC 822 является то, что в ситуации, когда несколько сообщений или записей в данном формате помещаются в файл, границы записей могут быть неочевидными — как лишенный интеллекта компьютер определит, где заканчивается неструктурированное текстовое тело сообщения и начинается следующий заголовок? Исторически сложились несколько различных соглашений для разграничения сообщений в почтовых ящиках. Старейший и наиболее широко поддерживаемый способ, при котором каждое сообщение начинается со строки, содержащей в начале слово "From " и сведения об отправителе, не подходит для других видов записей. Он также требует, чтобы строки в тексте сообщения,

начиная с " From ", разделялись (обычно с помощью символа >), а данная практика нередко приводит к путанице.

В некоторых почтовых системах используются разграничительные строки, состоящие из управляющих символов, появление которых в сообщениях маловероятно, например, последовательность нескольких символов ASCII 01 (control-A). Стандарт MIME обходит данную проблему путем явного указания в заголовке длины сообщения, однако такое решение является ненадежным и, весьма вероятно, потерпит неудачу, если сообщения когда-либо редактировались вручную. Несколько лучшим решением является стиль record-jar, описанный далее в настоящей главе.

Примеры использования формата RFC 822 можно найти в любом электронном почтовом ящике.

5.2.3. Формат Cookie-Jar

Формат cookie-jar используется программой

fortune(1) для собственной базы данных случайных цитат. Он подходит для записей, которые представляют собой просто блоки неструктурированного текста. В качестве разделителя записей в данном формате применяется символ новой строки, за которым следуют символы %% (или иногда символ новой строки с последующим символом %). В приведенном ниже примере (5.3) приведен фрагмент файла цитат почтовых подписей.Пример 5.3. Файл программы fortune

"Среди многих злодеяний английского правления в Индии жесточайшим история сочтет Акт обезоруживания всей нации."

-- Мохатма Ганди (Mohandas Gandhi), "Автобиография", стр. 446

%

Людям некоторых провинций строго воспрещается владеть любыми мечами, короткими мечами, луками, копьями, огнестрельным оружием или оружием любого другого типа. Владение излишним инвентарем усложняет сбор налогов и податей, а также подстрекает к бунтам.

-- Тойотоми Хидеоши (Toyotomi Hideyoshi), диктатор Японии, август 1588

%

"Одним из обычных способов, с помощью которых тираны без сопротивления достигали своих целей, является обезоруживание людей и возведение в ранг преступления владение оружием."

-- Судья Верховного суда Джозеф Стори (Joseph Story), 1840

Хорошая практика допускает использование пробела после символа % при поиске разделителей записей. Это помогает справляться с ошибками, связанными с редактированием вручную. Еще лучше использовать последовательность символов %% и игнорировать весь текст от %% до конца строки.

С самого начала разделителем в формате cookie-jar была последовательность %%\n. Я

искал нечто более очевидное, чем символ %. По существу, все после %% интерпретируется как комментарий (или, по крайней мере, я так это писал) Кен Арнольд.

Простой формат cookie-jar подходит для блоков текста, которые не имеют естественно упорядоченной, различимой структуры выше уровня слов или поисковых ключей, отличающихся от их текстового содержания.

5.2.4. Формат record-jar

Разделители записей формата cookie-jar хорошо сочетаются с метаформатом RFC 822 для записей, образующих формат, который в данной книге называется "record-jar". Иногда требуется текстовый формат, поддерживающий множественные записи с различным набором явных имен полей. В таком случае одним из наименее неожиданных и самым дружественным по отношению к пользователям является формат, пример которого представлен ниже (см. пример 5.4). Пример 5.4. Основные характеристики трех планет в формате record-jar

Planet: Mercury

Orbital-Radius: 57,910,000 km

Diameter: 4,880 km

Mass: 3.30e23 kg

%%

Planet: Venus

Orbital-Radius: 108,200,000 km

Diameter: 12,103.6. km

Mass: 4.869e24 kg

%%

Planet: Earth

Orbital-Radius: 149,600,000 km Diameter: 12,756.3. km

Mass: 5.972e24 kg

Moons: Luna

В качестве разделителя записей, несомненно, могла бы использоваться пустая строка. Однако строка, содержащая последовательность "%%\n", является более явной и вряд ли созданной в результате оплошности во время редактирования (два печатаемых символа лучше, чем один, поскольку их появление невозможно в результате одной опечатки). Хорошая практика в таком формате — просто игнорировать пустые строки.

Если записи имеют неструктурированную текстовую часть, то формат record-jar вплотную приближается к почтовому формату. В таком случае важно иметь четко определенный способ отделения разделителя записей, так чтобы данный символ мог содержаться в тексте. В противном случае считывающий код однажды "задохнется" на неверно сформированной

текстовой части. Ниже указываются некоторые методики, аналогичные заполнению байтами (byte-stuffing; описывается далее в данной главе).

Формат record-jar подходит для наборов связей "поле-атрибут", подобных DSV-стилю, однако имеет переменный состав полей и, возможно, связанный с ними неструктурированный текст.

5.2.5. XML

Язык XML представляет собой очень простой синтаксис, подобный HTML, — теги в угловых скобках и литеральные последовательности, начинающиеся с амперсанта. XML почти настолько же прост, насколько может быть простой разметка простого текста, а, кроме того, он позволяет выражать рекурсивно вложенные структуры данных. XML — только низкоуровневый синтаксис, для того чтобы снабдить его семантикой, необходимо определение типа документа (например, XHTML) и связанная логика приложений.

XML хорошо подходит для сложных форматов данных (для чего в Unix-традициях старой школы использовался бы формат подобный RFC 822, разделенный на строфы), хотя для более простых структур он является избыточным. Его особенно целесообразно использовать для форматов, содержащих сложную вложенную или рекурсивную структуру данных, которую метаформат RFC 822 не поддерживает должным образом. Книга

"XML in a Nutshell" [32] является хорошим введением при изучении данного формата.

Среди наибольших трудностей для правильного проектирования текстового файлового формата следует упомянуть проблемы использования кавычек, пробелов и других элементов низкоуровневого синтаксиса. Нестандартные файловые форматы нередко страдают от несколько недоработанного синтаксиса, который не полностью соответствует другим подобным форматам. Большинство данных проблем устраняется путем использования стандартного формата, такого как XML, который поддается контролю и позволяет осуществлять синтаксический анализ средствами стандартной библиотеки. Кит Паккард.

В примере 5.5. приведен простой образец конфигурационного файла на основе формата XML. Данный файл является частью инструмента

kdeprint , который поставляется с офисным пакетом поддерживаемой в Linux среды KDE с открытым исходным кодом. В нем описаны параметры для операции фильтрации изображений в PostScript и их преобразование в аргументы для команды фильтра. Другой информативный пример приведен в главе 8 при описании программы

Glade .

Преимуществом XML является то, что он часто позволяет обнаружить неверно сформированные, поврежденные или некорректно сгенерированные данные посредством проверки синтаксиса, даже "не зная" их семантики.

Наиболее серьезной проблемой формата XML является то, что он недостаточно хорошо обрабатывается традиционными инструментальными средствами Unix. Для считывания данного формата программе необходим синтаксический анализатор XML, а это означает использование громоздких, сложных программ. Кроме того, сам по себе XML является достаточно громоздким, из-за чего порой трудно найти данные среди всей разметки.

Одной прикладной областью, в которой XML, безусловно, выигрывает, являются форматы разметки для файлов документов (подробнее данная тема освещается в главе 18). Плотность

разметки в таких документах небольшая по сравнению с большими блоками простого текста, поэтому традиционные средства Unix довольно хорошо справляются с простыми операциями поиска и трансформации текста. Пример 5.5. XML-формат

```
&lt;?xml version="1.0"?&gt;
&lt;kprintfilter name="imagetops"&gt;
 &lt;filtercommand data="imagetops %filterargs %filterinput %filteroutput" /&gt;
 &lt;filterargs&gt;
  &lt;filterarg name="center" description="Image centering"
   format="-nocenter" type="bool" default="true"&gt;
   &lt;value name="true" description="Yes" /&gt;
   &lt;value name="false" description="No" /&gt;
  &lt;/filterarg&gt;
  &lt;filterarg name="turn"
   description="Image rotation" format="-%value" type="list" default="auto"&gt;
   &lt;value name="auto" description="Automatic" /&gt;
   &lt;value name="noturn" description="None" /&gt;
   &lt;value name="turn" description="90 deg" /&gt;
  &lt;/filterarg&gt;
  &lt;filterarg name="scale"
   description="Image scale" format="-scale %value" type="float"
   min="0.0" max="1.0" default="1.000" /&gt;
  &lt;filterarg name="dpi"
   description="Image resolution" format="-dpi %value"
   type="int" min="72" max="1200" default="300" /&gt;
 &lt;/filterargs&gt;
 &lt;filterinput&gt;
  &lt;filterarg name="file" format="%in" /&gt;
  &lt;filterarg name="pipe" format="" /&gt;
 &lt;/filterinput&gt;
 &lt;filteroutput&gt;
  &lt;filterarg name="file" format="&gt; %out" /&gt;
```

<filterarg name="file" format="" />

</filteroutput>

</kprintfilter>

Своеобразным мостом между этими мирами является формат PYX — строчно-ориентированное преобразование XML, которое можно обработать с помощью традиционных строчных текстовых средств Unix, а затем без потерь перевести обратно в XML. Web-поиск по ключевому слову "Pyxie" позволит найти ссылки на соответствующие ресурсы. Инструментальный набор xmltk движется в противоположном направлении, предоставляя потоковые средства, аналогичные

grep(1) и

sort(1) , для фильтрации XML-документов. Поиск по слову "xmltk" в Web поможет найти данный инструментарий.

XML может упрощать или, напротив, усложнять конструкцию. Он окружен активной рекламой, однако не стоит становиться жертвой моды, безоговорочно принимая или отвергая данный формат. Выбирать следует осторожно, руководствуясь принципом KISS.

5.2.6. Формат Windows INI

Многие программы в Microsoft Windows используют текстовый формат данных, подобный фрагменту, приведенному в примере 5.6. В данном примере необязательные ресурсы с именами account, directory, numeric_id и developer связываются с именованными проектами python, sng, fetchmail и py-howto. В записи DEFAULT указаны значения, которые используются в случае, если они не предоставляются именованными записями. Пример 5.6. Формат Windows INI

```
[DEFAULT]

account = esr


[python]

directory = /home/esr/cvs/python/

developer = 1


[sng]

directory = /home/esr/WWW/sng/

numeric_id = 1012

developer = 1
```

[fetchmail]

numeric_id = 18364

[py-howto]

account = eric

directory = /home/esr/cvs/py-howto/

developer = 1

Такой стиль формата файлов данных не характерен для операционной системы Unix, однако некоторые Linux-программы (особенно Samba, пакет средств доступа к Windows-файлам из Linux) под влиянием Windows поддерживают его. Данный формат является четким и неплохо спроектированным, однако, как и в случае XML,

grep(1) или традиционные средства сценариев Unix не обрабатывают его должным образом.

.INI-формат целесообразно использовать, если данные естественным образом соответствуют его двухуровневой организации пар "имя-атрибут", собранных в группы в именованных записях или секциях. Он плохо подходит для данных с полностью рекурсивной древовидной структурой (для этого лучше подходит XML), и является избыточным для простого списка связей "имя-значение" (в этом случае лучше использовать DSV).

5.2.7. Unix-соглашения по текстовым файловым форматам

Существуют давние традиции Unix, определяющие вид текстовых форматов данных. Большинство из них происходит от одного или нескольких описанных выше стандартных метаформатов Unix. Разумно следовать данным соглашениям, если нет весомых и специфических причин поступать иначе.

В главе 10 рассматривается другой набор соглашений, применяемых для файлов конфигурации программ, однако, следует заметить, что в нем используются некоторые из описанных выше правил (особенно касающиеся лексического уровня, т.е. правила, согласно которым символы собираются в лексемы).

•

Если возможно, следует включать одну запись в строку, ограниченную символом новой строки. Такой подход упрощает извлечение записей с помощью средств обработки текстовых потоков. Для взаимного обмена данными с другими операционными системами разумно сделать работу синтаксического анализатора данного файлового формата независимой от того, завершается ли строка символом LF или символами CR-LF. Кроме того, в таких форматах завершающие пробелы традиционно игнорируются, что защищает от распространенных ошибок редакторов.

•

Если возможно, следует использовать менее 80 символов в строке. Это позволит просматривать данные в терминальном окне обычного размера. Если многие записи должны

содержать более 80 символов, то следует обратить внимание на формат с использованием строф (см. ниже).

•

Использовать символ

#

как начало комментария . Полезно иметь способ внедрения замечаний и комментариев в файлы данных. Еще лучше, если они фактически являются частью структуры файла и поэтому будут сохраняться инструментами, которые распознают формат данного файла. Для комментариев, которые не сохраняются во время синтаксического анализа, знак # является стандартным начальным символом.

•

Следует поддерживать соглашение об использовании обратной косой черты. Наименее неожиданный способ поддержки непечатаемых управляющих символов заключается в синтаксическом анализе escape-последовательностей с обратной косой чертой в стиле C: \n для разделителя строк, \r для возврата каретки, \t для табуляции, \b для возврата на один символ назад, \f для разделителя страниц, \e для ASCII-символа escape (27), \nnn или \onnn, или \0nnn для символа с восьмеричным значением nnn, \xnn для символа с шестнадцатеричным значением nn, \dnnn для символа с десятичным значением nnn, \\ для буквального использования обратной косой черты. В более новом, но заслуживающем внимания соглашении последовательность \unnnn используется для шестнадцатеричного Unicod-литерала.

•

В форматах с использованием одной строки для одной записи в качестве разделителя полей следует применять двоеточие или серию пробелов. Соглашение об использовании двоеточия, вероятно, возникло вместе с файлом паролей Unix. Если поля должны содержать экземпляры разделителя (или разделителей), то следует использовать обратную косую черту как префикс для буквального представления этих символов.

•

Не следует делать важными различия между символами табуляции и пробелами. Это может привести к серьезным проблемам, в случае если настройки табуляции в пользовательских редакторах отличаются. В более общем смысле они препятствуют правильному зрительному восприятию. Использование одного символа табуляции в качестве разделителя полей особенно чревато возникновением проблем. С другой стороны, хорошая практика допускает использование любой последовательности символов табуляции и пробелов в качестве разделителя полей.

•

Шестнадцатеричное представление предпочтительнее восьмеричного. Шестнадцатеричные пары и четверки проще для зрительного преобразования в байты и современные 32- и 64-битовые слова, чем восьмеричные цифры, состоящие из трех битов каждая. Кроме того, этот подход несколько более эффективен. Данное правило необходимо подчеркнуть, поскольку некоторые старые средства Unix, такие как утилита

od(1) , нарушают его. Это наследие связано с размерами полей команд в машинных языках для давних мини-компьютеров PDP.

•

Для сложных записей рекомендуется использовать формат со "строфами": несколько строк в записи, причем записи разделяются строкой, состояний из последовательности

%%\n

или

%\n

. Разделители создают удобные видимые границы для визуального контроля файла.

•

В форматах со строфами следует использовать либо одно поле записи на строку, либо формат записей, подобный заголовкам электронной почты RFC 822, где поле начинается с отделенного двоеточием ключевого слова (названия поля). Второй вариант целесообразно использовать, когда поля часто либо отсутствуют, либо содержат более 80 символов, или когда плотность записей невысока (например, часто встречаются пустые поля).

•

В форматах со строфами следует обеспечивать поддержку продолжения строк. В ходе интерпретации файла необходимо либо игнорировать обратную косую черту с последующим пробелом, либо интерпретировать разделитель строк с последующим пробелом эквивалентно одному пробелу так, чтобы длинная логическая строка могла быть свернута в короткие (легко редактируемые) физические строки. Также существует соглашение, рекомендующее игнорировать завершающие пробелы в таких форматах. Данное соглашение защищает от распространенных ошибок редакторов.

•

Рекомендуется либо включать номер версии, либо разрабатывать формат в виде самоописательных независимых друг от друга блоков. Если существует даже минимальная вероятность того, что потребуется вносить изменения в формат или расширять его, необходимо включить номер версии, с тем чтобы код мог правильно обрабатывать все версии. В качестве альтернативы следует проектировать формат, состоящий из самоописательных блоков данных, так чтобы можно было добавить новые типы блоков без нарушения прежнего кода.

•

Рекомендуется избегать проблем, вызванных округлением чисел с плавающей точкой. В процессе преобразования чисел с плавающей точкой из двоичного в текстовый формат и обратно может быть потеряна точность в зависимости от качества используемой библиотеки преобразования. Если структура, которая подвергается маршалингу/демаршалингу, содержит числа с плавающей точкой, то следует протестировать преобразование в обоих направлениях. Если преобразование в каком- либо направлении сопряжено с ошибками округления, то необходимо предусмотреть вариант сохранения поля с плавающей точкой в необработанном двоичном виде или кодировать его как текстовую строку. Если программа пишется на языке C или каком-либо другом, имеющем доступ к функциям C printf/scanf, то данную проблему можно разрешить с помощью спецификатора C99 %a.

•

Не следует сжимать или кодировать в двоичном виде только часть файла. См. ниже.

5.2.8. Аргументы "за" и "против" сжатия файлов

Во многих современных Unix-проектах, таких как OpenOffice.org и AbiWord, в настоящее время в качестве формата файлов данных используется XML, сжатый с помощью программ

zip(1) или

gzip(1) . Сжатый XML комбинирует экономию пространства с некоторыми преимуществами текстового формата — в особенности он позволяет избежать проблемы двоичных форматов, состоящей в том, что в них необходимо выделение пространства для информации, которая может не использоваться в особых случаях (например, для необычных опций или больших диапазонов). Однако по этому поводу еще ведутся споры, и в связи с этим идет поиск компромиссов, обсуждение которых представлено в данной главе.

С одной стороны, эксперименты показывают, что документы в сжатом XML- файле обычно значительно меньше по размеру, чем собственный файловый формат программы Microsoft Word, двоичный формат, который на первый взгляд занял бы меньше места. Причина связана с фундаментальным принципом философии Unix: решать одну задачу хорошо. Создание отдельного средства для качественного выполнения компрессии является более эффективным, чем специальное сжатие частей файла, поскольку такое средство может просмотреть все данные и использовать

все повторения в них.

Кроме того, путем отделения формы представления от используемого специфического метода сжатия, разработчик оставляет открытой возможность использования в будущем других методов компрессии с минимальными изменениями синтаксического анализа файлов, а возможно, даже без изменений.

С другой стороны, сжатие несколько вредит прозрачности. В то время как человек способен по контексту оценить, возможно ли путем декомпрессии данного файла получить какую-либо полезную информацию, то средства, подобные

file(1), по состоянию на середину 2003 года все еще не могут анализировать упакованные файлы.

Некоторые специалисты склоняются к менее структурированному формату сжатия — непосредственно сжатые программой

gzip(1) XML-данные, например, без внутренней структуры и самоидентифицирующего заголовочного блока, обеспеченного утилитой

zip(1) . Наряду с тем, что использование формата, подобного

zip(1) , решает проблему идентификации, оно также означает, что декодирование таких файлов будет сложным для программ, написанных на простых языках сценариев.

Любое из описанных решений (чистый текст, чистый двоичный формат или сжатый текст) может быть оптимальным в зависимости от того, какое значение разработчик придает экономии дискового пространства, воспринимаемости или максимальной простоте при написании средств просмотра. Суть предшествующего изложения заключается не в пропаганде какого-либо из описанных подходов, а скорее в предложении способов четкого анализа вариантов и компромиссов проектирования.

Это означает, что истинным решением в духе Unix было бы, возможно, настроить утилиту

file(1) для просмотра префиксов сжатых файлов, а в случае неудачи, написание в оболочке сценария-упаковщика вокруг

file(1) , который с помощью программы

gunzip(1) распаковывал бы сжатый файл для просмотра.

5.3. Проектирование протоколов прикладного уровня

В главе 7 рассматриваются преимущества разбиения сложных приложений на взаимодействующие процессы, которые обмениваются друг с другом данными посредством специфичного для них набора команд или протокола. Преимущества использования текстовых форматов файлов данных также характерны для этих специфичных для приложений протоколов.

Если протокол уровня приложения является текстовым и может быть проанализирован визуально, то многое становится проще. Сильно упрощается интерпретация "распечаток" транзакций, а также написание тестовых программ.

Серверные процессы часто запускаются управляющими программами, подобными демону

inetd(8) , так что сервер получает команды на стандартный ввод и отправляет ответы на стандартный вывод. Данная модель "CLI-сервера" подробнее описана в главе 11.

CLI-сервер с набором команд простой конструкции имеет то ценное свойство, что тестирующий его человек для проверки работы программного обеспечения может вводить команды непосредственно в серверный процесс.

Также необходимо учитывать принцип сквозного (end-to-end) проектирования. Каждому разработчику протоколов следует прочесть классический документ

"End-to-End Arguments in System Design" [73]. Часто возникают серьезные вопросы о том, на каком уровне набора протоколов следует поддерживать такие функции, как безопасность и аутентификация. В указанной статье приводятся некоторые хорошие концептуальные средства для анализа. Третьей проблемой является проектирование высокопроизводительных протоколов прикладного уровня. Более подробно данная проблема освещается в главе 12.

До 1980 года традиции проектирования протоколов прикладного уровня в Internet развивались отдельно от операционной системы Unix[53]. Однако с 80-х годов эти традиции полностью прижились в Unix.

Internet-стиль иллюстрируется в данной главе на примере трех протоколов прикладного уровня, которые входят в число наиболее интенсивно используемых и рассматриваются в среде Internet-хакеров как принципиальные: SMTP, POP3 и IMAP. Все три определяют различные аспекты передачи почты (одной из двух наиболее важных прикладных задач сети наряду с World Wide Web), однако решаемые ими проблемы (передача сообщений, установка удаленного состояния, указание ошибочных условий) также являются характерными для непочтовых протоколов и обычно решаются с помощью подобных методик.

5.3.1. Учебный пример: SMTP, простой протокол передачи почты

В примере 5.7. иллюстрируется транзакция SMTP (Simple Mail Transfer Protocol — простой протокол передачи почты), который описан в спецификации RFC 2821. В данном примере строки, начинающиеся с

C:, отправляются почтовым транспортным агентом (Mail Transport Agent — MTA), который отправляет почту, а строки, начинающиеся с

S: , возвращаются агентом (MTA), принимающим ее. Текст,

выделенный курсивом , представляет собой комментарии и не является частью реальной транзакции.

Так почта передается между Internet-машинами. Следует отметить ряд особенностей: формат команд и аргументов запросов, ответы, содержащие код состояния, за которым следует информационное сообщение, и то, что полезная нагрузка команды DATA ограничивается строкой, содержащей одну точку. Пример 5.7. SMTP-сеанс

C:

<клиент подключается к служебному порту 25>

C: HELO snark.thyrsus.com

отправляющий узел

идентифицирует себя

S: 250 OK Hello snark, glad to meet you

подтверждение получателя

C: MAIL FROM: <esr@thyrsus.com>

идентификация отправляющего

пользователя

S: 250 <esr@thyrsus.com>... Sender ok

подтверждение получателя

C: RCPT TO: cor@cpmy.com

идентификация целевого

пользователя

S: 250 root... Recipient ok

подтверждение получателя

C: DATA

S: 354 Enter mail, end with "." on a line by itself

C: Звонил Scratch. Он хочет снять с нами

C: комнату в Balticon.

C: .

отправляется окончание

многострочной записи

S: 250 WAA01865 Message accepted for delivery

C: QUIT

отправитель отключается

S: 221 cpmy.com closing connection

получатель отключается

C:

<клиент разрывает соединение>

SMTP один из двух или трех старейших протоколов прикладного уровня, которые до сих пор используются в Internet. Он прост, эффективен и выдержал проверку временем. Особенности, описанные здесь, часто повторяются в других Internet-протоколах. Если существует какой-либо один образец того, как выглядит хорошо спроектированный протокол Internet-приложения, то им, несомненно, является SMTP.

5.3.2. Учебный пример: POP3, почтовый протокол 3-й версии

Другим классическим Internet-протоколом является POP3 (Post Office Protocol — почтовый протокол 3-й версии). Он также используется для транспортировки почты, но если SMTP является "толкающим" протоколом с транзакциями, инициированными отправителем почты, то POP3 является протоколом "тянущим", а его транзакции инициируются получателем почты. Internet-пользователи с непостоянным доступом (например, по коммутируемым соединениям) могут накапливать свою почту на почтовом сервере, а затем, подключившись к POP3-серверу, получать почту на персональные машины.

В примере 5.8. показан POP3-сеанс. В данном примере строки, начинающиеся с

C: , отправляются клиентом, а строки, начинающиеся с

S: , почтовым сервером. Необходимо отметить множество моментов, сходных с SMTP. Протокол POP3 также является текстовым и строчно-ориентированным. В данном случае, так же как в случае SMTP, отправляются блоки полезной нагрузки сообщений, ограниченные строкой, содержащей одну точку, за которой следует ограничитель строки, и даже используется такая же команда выхода — QUIT. Подобно SMTP, в протоколе POP3 каждая клиентская операция подтверждается ответной строкой, которая начинается с кода состояния и включает в себя информационное сообщение, понятное человеку.Пример 5.8. POP3-сеанс

C:

<клиент подключается к служебному порту 110>

S: +OK POP3 server ready <1896.697l@mailgate.dobbs.org>

C: USER bob

S: +OK bob

C: PASS redqueen

S: +OK bob's maildrop has 2 messages (320 octets)

C: STAT

S: +OK 2 320

C: LIST

S: +OK 2 messages (320 octets)

S: 1 120

S: 2 200

S: .

C: RETR 1

S: +OK 120 octets

S: <POP3-сервер отправляет текст сообщения 1>

S: .

C: DELE 1

S: +OK message 1 deleted

C: RETR 2

S: +OK 200 octets

S: <POP3-сервер отправляет текст сообщения 2>

S: .

C: DELE 2

S: +OK message 2 deleted

C: QUIT

S: +OK dewey POP3 server signing off (maildrop empty)

C: <клиент разрывает соединение>

В то же время существует несколько отличий. Наиболее очевидным из них является то, что в протоколе POP3 используются маркеры состояния вместо трехзначных кодов состояния, применяемых в SMTP. Несомненно, семантика запросов различна, однако "семейное

сходство" (которое более подробно будет описано далее в настоящей главе при рассмотрении общего метапротокола Internet) очевидно.

5.3.3. Учебный пример: IMAP, протокол доступа к почтовым сообщениям

Чтобы завершить рассмотрение примеров с протоколами прикладного уровня Internet, рассмотрим протокол IMAP (Internet Message Access Protocol — протокол доступа к почтовым сообщениям в Internet). IMAP — другой почтовый протокол, спроектированный в несколько ином стиле. Как и в предыдущих примерах, строки, начинающиеся с

C:, отправляются клиентом, а строки, начинающиеся с 5:, отправляются почтовым сервером. Текст,

выделенный курсивом , представляет собой комментарии и не является частью реальной транзакции.Пример 5.9. IMAP-сеанс

C:

<клиент подключается к служебному порту 143>

S: * OK example.com IMAP4rev1 V12.264 server ready

C: A0001 USER "frobozz" "xyzzy"

S: * OK User frobozz authenticated

C: A0002 SELECT INBOX

S: * 1 EXISTS

S: * 1 RECENT

S: * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)

S: * OK [UNSEEN 1] first unseen message in /var/spool/mail/esr

S: A0002 OK [READ-WRITE] SELECT completed

C: A0003 FETCH 1 RFC822.SIZE

получение размеров сообщений

S: * 1 FETCH (RFC822.SIZE 2545)

S: A0003 OK FETCH completed

C: A0004 FETCH 1 BODY[HEADER]

получение заголовка первого

сообщения

S: * 1 FETCH (RFC822.HEADER {1425}

<сервер отправляет 1425 октетов полезной нагрузки сообщения>

S:)

S: A0004 OK FETCH completed

C: A0005 FETCH 1 BODY[TEXT]

получение тела первого

сообщения

S: * 1 FETCH (BODY[TEXT] {1120}

<сервер отправляет 1120 октетов полезной нагрузки сообщения>

S:)

S: * 1 FETCH (FLAGS (\Recent \Seen))

S: A0005 OK FETCH completed

C: A0006 LOGOUT

S: * BYE example.com IMAP4rev1 server terminating connection

S: A0006 OK LOGOUT completed

C

: <клиент разрывает соединение>

В IMAP полезная нагрузка ограничивается несколько иначе. Вместо завершения блока полезной нагрузки с помощью точки перед ним отправляется его длина. Это несколько увеличивает накладные расходы на сервере (сообщения должны быть скомпонованы заранее, их невозможно просто установить в поток после того, как отправка инициирована), однако упрощает работу клиента, поскольку предоставляет возможность заранее определить объем пространства, которое необходимо выделить в целях буферизации сообщения для его обработки в целом.

Кроме того, следует заметить, что каждый ответ маркируется последовательной меткой, передаваемой в запросе. В данном примере такие метки имеют форму A000n, однако клиент может генерировать любой маркер в данном поле. Данная особенность позволяет направлять серверу поток IMAP-команд, не ожидая ответов. Конечный автомат клиента может затем просто интерпретировать ответы и блоки полезной нагрузки по мере их возвращения. Данная методика сокращает задержку.

Протокол IMAP (который был разработан для замены POP3) является превосходным образцом продуманной и мощной конструкции прикладного протокола в Internet, примером, достойным изучения и подражания.

5.4. Метаформаты протоколов прикладного уровня

Подобно тому, как были усовершенствованы метаформаты файлов данных, чтобы упростить сериализацию для хранения этих данных, метаформаты протоколов прикладного уровня были усовершенствованы, чтобы упростить сериализацию для передачи данных через сети.

Правда, ввиду того, что полоса пропускания сети является более дорогой, чем устройства хранения, экономичность транзакций приносит больший выигрыш. Однако преимущества прозрачности и способности к взаимодействию текстовых форматов являются достаточно устойчивыми, поэтому большинство проектировщиков не поддались искушению оптимизировать производительность ценой читабельности.

5.4.1. Классический метапротокол прикладного уровня в Internet

RFC 3117 Маршала Роуза (Marshall Rose),

"On the Design of Application Protocols" [54] представляет исключительный обзор вопросов проектирования протоколов прикладного уровня в Internet. В данном документе проясняются несколько черт классических протоколов прикладного уровня Internet, которые были отмечены выше при изучении SMTP, POP и IMAP, а также предоставляется информативная классификация таких протоколов. Данный документ входит в число рекомендуемой литературы.

Классический метапротокол Internet является текстовым. В нем используются однострочные запросы и ответы, за исключением блоков полезной нагрузки, которые могут содержать множество строк. Блоки полезной нагрузки отправляются либо с предшествующей длиной, выраженной в октетах, либо с ограничителем, который представляет собой строку ".\r\n". В последнем случае полезная нагрузка

заполняется байтами . Все строки, начинающиеся с точки, дополняются впереди еще одной точкой, а получатель отвечает за опознание ограничителя и удаление заполнения. Строки ответов состоят из кода состояния, за которым следует удобочитаемое сообщение.

Абсолютным преимуществом данного классического стиля является то, что его просто расширять. Структура синтаксического анализа и конечного автомата не нуждается в серьезных изменениях, для того чтобы приспособиться к новым запросам. И поэтому очень просто можно программировать реализации, которые способны осуществлять синтаксический анализ неизвестных запросов и возвращать ошибку или игнорировать их. Все протоколы SMTP, POP3 и IMAP за время их существования довольно часто незначительно расширялись с минимальными проблемами взаимодействия. В противоположность им, примитивно спроектированные двоичные протоколы печально известны как неустойчивые.

5.4.2. HTTP как универсальный протокол прикладного уровня

С тех пор как приблизительно в 1993 году World Wide Web достигла критической массы, проектировщики прикладных протоколов демонстрируют усиливающуюся тенденцию к размещению специализированных протоколов над HTTP, используя Web-серверы как общие служебные платформы.

Такая стратегия жизнеспособна, поскольку на уровне транзакций HTTP является весьма простым и общим протоколом. HTTP-запрос представляет собой сообщение в формате, подобном RFC-822/MIME. Как правило, заголовки содержат идентификационную информацию и сведения по аутентификации, а первая строка представляет собой вызов метода на определенном ресурсе, указанном с помощью универсального указателя ресурсов (Universal

Resource Indicator — URI). Наиболее важными методами являются GET (доставка ресурса), PUT (модификация ресурса) и POST (отправка данных в форму или серверному процессу). Наиболее важной формой URI является URL, или Uniform Resource Locator (унифицированный указатель ресурса), который идентифицирует ресурс по типу службы, имени узла и расположению ресурса на данному узле. HTTP-ответ является простым RFC-822/MIME-сообщением и может вмещать в себе произвольное содержимое, которое интерпретируется клиентом.

Web-серверы управляют транспортным уровнем и уровнем мультиплексирования запросов HTTP, а также стандартными типами служб, таких как http и ftp. Сравнительно просто писать для Web-серверов дополнительные модули, которые обрабатывают нестандартные типы служб, а также осуществлять диспетчеризацию по другим элементам формата URI.

Кроме того, что данный метод позволяет избежать большого количества низкоуровневых деталей, он также означает, что протокол прикладного уровня образует туннель через стандартный порт HTTP-службы и не нуждается в собственном TCP/IP-порте. Это можно рассматривать как явное преимущество. Большинство брандмауэров оставляют порт 80 открытым, однако попытки пробиться через другие порты могут быть чреваты как техническими трудностями, так и теми, что связаны с политикой.

Данное преимущество сопряжено с некоторым риском. Это означает, что возрастает сложность Web-сервера и его дополнительных модулей, и взлом какого-либо кода может иметь серьезные последствия, связанные с безопасностью. Может усложниться изоляция и отключение проблемных служб. В данном случае целесообразны обычные компромиссы между безопасностью и удобством.

В RFC 3205,

"On the Use of HTTP As a Substrate" [55] приведены хорошие рекомендации по проектированию, касающиеся использования протокола HTTP в качестве нижнего уровня для протокола приложения, включая обобщение связанных компромиссов и проблем.

5.4.2.1. Учебный пример: база данных CDDB/freedb.org

Аудио компакт-диски (CD) содержат последовательность музыкальных записей в цифровом формате, который называется CDDA-WAV. Они были разработаны для проигрывания на очень простых бытовых электронных устройствах за несколько лет до того, как универсальные компьютеры стали развивать чистую скорость и звуковые возможности, достаточные для декодирования записей налету. Поэтому в данном формате нет запаса даже для хранения простой метаинформации, такой как названия альбомов и записей. Однако в современных компьютерных проигрывателях компакт-дисков данная информация обязательно должна быть предусмотрена, с тем чтобы пользователи могли составлять и редактировать списки воспроизведения.

В Internet существует по крайней мере два репозитория, предоставляющих преобразование между хэш-кодом, который вычисляется по таблице длины записей на компакт-диске, и записями, содержащими имя музыканта/название альбома/название записи. Первоначальным сайтом был cddb.org, однако существует другой сайт, freedb.org, который, вероятно, в настоящее время является наиболее полным и широко используемым. Оба сайта полагаются на своих пользователей в решении тяжелейшей задачи по поддержанию актуального состояния базы данных по мере выхода новых компакт-дисков. Сайт freedb.org возник как протест разработчиков после того, как CDDB приняла решение о частной

собственности на всю информацию, собранную пользователями.

Запросы к данным службам могли бы быть реализованы в форме частного протокола прикладного уровня на поверхности TCP/IP. Однако в таком случае потребовались бы такие мероприятия, как получение нового выделенного TCP/IP-порта и создание канала через тысячи брандмауэров. Вместо этого данная служба реализована над HTTP как простой CGI-запрос (как будто хэш-код компакт-диска вводится при заполнении пользователем Web-формы).

Такой выбор предоставляет всей существующей инфраструктуре библиотек HTTP и Web-доступа в различных языках программирования возможность поддерживать программы для запроса информации и обновления этой базы данных. В результате добавление такой поддержки к программным проигрывателям компакт-дисков является почти тривиальной задачей, и фактически все программные проигрыватели способны использовать упомянутые базы данных.

5.4.2.2. Учебный пример: протокол IPP

IPP (Internet Printing Protocol — протокол печати через Internet) является удачным, широко распространенным стандартом для управления принтерами, доступными через сеть. Указатели на RFC, реализации и многие другие связанные материалы доступны на сайте рабочей группы "Printer Working Group", подразделения IETF <http://www.pwg.org/ipp/>.

В протоколе IPP в качестве транспортного уровня используется HTTP 1.1. Все IPP-запросы проходят через вызов POST-метода HTTP, а ответы являются обычными HTTP-ответами. (В разделе 4.2. RFC 2568, "

Rationale for the Structure of the Model and Protocol for the Internet Printing Protocol" данный выбор превосходно обосновывается. Указанный раздел заслуживает изучения разработчиками новых протоколов прикладного уровня.)

Что же касается программного обеспечения, то широко распространен протокол HTTP 1.1. Он уже решает множество проблем транспортного уровня, которые в противном случае отвлекали бы конструкторов и создателей протоколов от семантики печати. Существует возможность простого расширения данного протокола, поэтому вполне реальной представляется перспектива роста IPP. Модель CGI-программирования для обработки POST-запросов понятна, а инструменты для разработки широко доступны.

Большинство сетевых принтеров уже имеют встроенный Web-сервер, поскольку в этом состоит естественный путь предоставления пользователям возможности удаленно запрашивать сведения о состоянии принтера. Таким образом, инкрементная стоимость добавления IPP-службы в программно-аппаратное обеспечение принтера невысока. (Данный аргумент применим к чрезвычайно широкому диапазону другого сетевого аппаратного обеспечения, включая торговые автоматы и кофеварки[56].)

Единственный серьезный недостаток расположения IPP над HTTP заключается в том, что IPP полностью управляется клиентскими запросами. Поэтому в данной модели отсутствует пространство для отправки принтерами асинхронных извещений обратно клиентам. (Однако более интеллектуальные клиенты могли бы запускать примитивный HTTP-сервер для получения таких извещений, отформатированных в виде HTTP-запросов от принтера.)

5.4.3. BEEP: Blocks Extensible Exchange Protocol

BEEP (ранее BXXP), протокол для расширяемого обмена блоками информации является общим протокольным аппаратом, который конкурирует с HTTP в качестве универсального нижнего уровня для протоколов прикладного уровня. Существует открытая ниша, поскольку до сих пор нет другого, заслуживающего большего доверия, метапротокола, пригодного для действительно одноранговых приложений, как противоположности клиент-серверным приложениям, с которыми хорошо справляется HTTP. На сайте проекта <http://www.beepcore.org/beepcore/docs/sl-beep.jsp> предоставляется доступ к стандартам и реализациям с открытым исходным кодом на нескольких языках.

Протокол BEEP обладает функциями для поддержки как клиент-серверного, так и одноранговой режимов. Создатели BEEP спроектировали протокол и библиотеку поддержки таким образом, что выбор верных параметров избавляет от запутанных проблем, таких как кодировка данных, управление потоком, обработка перегрузок, поддержка сквозного шифрования и компоновка большого ответа, составленного из множества передач.

BEEP-узлы обмениваются между собой последовательностями самоописательных двоичных пакетов, которые подобны типам блоков в PNG. Данная конструкция более приспособлена к экономии и менее к прозрачности, чем классические Internet-протоколы или HTTP, и может быть наилучшим выбором при необходимости передавать большие объемы данных. Протокол BEEP также позволяет избежать проблемы HTTP, которая заключается в том, что все запросы должны быть инициированы клиентом. Это преимущество проявляется в ситуациях, когда серверу необходимо отправлять асинхронные извещения о состоянии обратно клиенту.

На момент написания книги (середина 2003 года) BEEP все еще является новой технологией и имеет только несколько демонстрационных проектов. Однако статьи по BEEP представляют собой хорошие аналитические обзоры лучшей практики в проектировании протоколов. Даже если сам по себе протокол BEEP не получит широкого признания, эти статьи в качестве учебных материалов, надолго сохранят свою ценность.

5.4.4. XML-RPC, SOAP и Jabber

В проектировании прикладных протоколов усиливается тенденция к использованию XML внутри MIME для структурирования запросов и блоков полезной нагрузки. BEEP-узлы используют данный формат для согласования каналов. По пути развития XML движутся три основных протокола: XML-RPC и SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) для реализации удаленного вызова процедур и Jabber для обмена мгновенными сообщениями. Все три протокола представляют собой типы XML-документов.

XML-RPC весьма выдержан в духе Unix (его автор отмечает, что он начал изучать программирование в 1970-х годах, читая оригинальный исходный код Unix). Подход к разработке данного протокола был осознанно минималистским. И тем не менее, протокол является весьма мощным. Он предоставляет способ для значительного большинства RPC-приложений, которые могут работать, распространяя скалярные булевы/целые/плавающие/строковые типы данных, выполнять их функции способом, простым для понимания и мониторинга. Онтология типов XML-RPC богаче онтологии текстовых потоков, однако остается простой и достаточно переносимой, для того чтобы функционировать в качестве ценной проверки сложности интерфейса. Существуют

реализации данного протокола с открытым исходным кодом. Ссылки на них, а также на соответствующие спецификации доступны на домашней странице XML-RPC <http://www.xmlrpc.com/>.

SOAP является более тяжеловесным RPC-протоколом с более развитой онтологией типов, которая включает в себя массивы и C-подобные структуры. Его создателей вдохновил XML-RPC, однако он заслуженно был назван "перепроектированной жертвой эффекта второй системы". К середине 2003 года работы по стандарту SOAP еще велись, однако пробная реализация в Apache остается черновой. Клиентские модули с открытыми исходными кодами на языках Perl, Python, Tcl и Java можно быстро найти с помощью Web-поиска. Проектная спецификация консорциума W3C доступна на странице <http://www.w3.org/TR/SOAP>.

Протоколы XML-RPC и SOAP, рассмотренные как методы удаленного вызова процедур, имеют некоторый связанный риск, который обсуждается в конце главы 7.

Jabber — одноранговый протокол, разработанный для поддержки мгновенного обмена сообщениями и присутствия. Он интересен как прикладной протокол тем, что поддерживает распространение XML-форм и интерактивных документов. Спецификации, документация и реализации с открытыми исходными кодами доступны на сайте организации Jabber Software Foundation <http://www.jabber.org/about/overview.html>.

6

Прозрачность: да будет свет

Красота в вычислениях более важна, чем в любой другой области технологии, поскольку программное обеспечение очень сложное. Красота — основная защита против сложности.

Machine Beauty: Elegance and the Heart of Technology (1998) —Дэвид Гелентер (David Gelernter)

В предыдущей главе акцентировалось внимание на важности текстовых форматов данных и протоколов прикладного уровня, т.е. тех форм представления, которые просты для изучения и взаимодействия. Они поддерживают качества конструкции, которые высоко ценятся в традиции Unix, но подробно обсуждаются в очень редких случаях (если вообще обсуждаются):

прозрачность и

воспринимаемость .

Программные системы являются прозрачными в том случае, когда они не имеют неясных мест или скрытых деталей. Прозрачность — пассивное качество. Программа прозрачна, если существует возможность сформировать простую ментальную модель ее поведения, которое фактически является предсказуемым во всех или большинстве случаев, поскольку изучая вопрос именно в аспекте механизма обработки данных, можно понять, что фактически происходит.

Программные системы являются воспринимаемыми, когда они включают в себя функции, которые способствуют мысленному построению корректной ментальной модели того, что делают данные системы и каким образом они работают. Так, например, хорошая

документация повышает воспринимаемость для пользователя, а хороший выбор переменных и имен функций повышает воспринимаемость для программиста. Воспринимаемость является активным качеством. Для того чтобы ее достичь, недостаточно просто отказаться от создания непонятных программ, необходимо изо всех сил стараться создавать удобные программы[57].

Прозрачность и воспринимаемость важны как для пользователей, так и для разработчиков программного обеспечения. Однако важность этих качеств проявляется по-разному. Пользователям нравятся эти свойства в пользовательском интерфейсе, поскольку они означают более простой процесс обучения. Прозрачность и воспринимаемость в данном случае являются большой частью того, что подразумевают пользователи, когда упоминают об "интуитивно понятном" пользовательском интерфейсе. Остальное в основном сводится к правилу наименьшей неожиданности. Свойства, которые делают пользовательские интерфейсы приятными и эффективными, более подробно обсуждаются в главе 11.

Разработчикам программного обеспечения нравятся данные качества в самом коде (т.е. в той части, которая не видна пользователям), поскольку им часто требуется понимать код достаточно хорошо, для того чтобы модифицировать и отлаживать его. Кроме того, программа, разработанная так, что разобраться в ее внутренних потоках данных несложно, более вероятно, является программой, которая не отказывает из-за некорректного обмена данными, незамеченного проектировщиком. Кроме того, такую программу, вероятнее всего, можно будет изящно развивать (включая внесение изменений, позволяющих приспособиться к ней новым кураторам, принявшим эстафету).

Прозрачность является главным компонентом того, что Дэвид Гелентер называет "красотой". Unix-программисты для упоминаемого Гелентером качества часто используют более специфический термин, заимствованный у математиков, — "изящество". Изящество является комбинацией мощности и простоты. Изящный код выполняет большую работу при малых затратах. Изящный код не просто корректен — его корректность очевидна и

прозрачна. Он не просто связывает алгоритм с компьютером, но также формирует понимание и уверенность в сознании того, кто его читает. В поисках изящества программисты создают лучший код. Изучение методики написания прозрачного кода является первым, значительным шагом к изучению того, как создавать изящный код, а забота о том, чтобы сделать код воспринимаемым, позволяет узнать, как сделать его прозрачным. Элегантный код характеризуется обоими качествами, как прозрачностью, так и воспринимаемостью.

Возможно, проще оценивать различие между прозрачностью и воспринимаемостью с помощью двух противоположных примеров. Исходный код ядра операционной системы Linux является в высшей степени прозрачным (принимая во внимание значительную сложность выполняемых задач). Вместе с тем, он совсем не является воспринимаемым — овладеть минимальными знаниями, необходимыми для работы с данным кодом, и понять особый язык его разработчиков трудно. Однако как только знания и понимание придут, все обретет смысл [58]. С другой стороны, библиотеки Emacs Lisp воспринимаемы, но не прозрачны. Овладеть достаточным набором знаний для настройки одного компонента просто, но постичь всю систему весьма сложно.

В данной главе рассматриваются особенности конструкций в Unix, которые поддерживают прозрачность и воспринимаемость не только в пользовательских интерфейсах, но и в тех частях программ, которые обычно не видны пользователям. В главе формулируется несколько полезных правил, которые можно применять в практическом программировании и разработке. Далее, в главе 19, описано, каким образом хорошая практика подготовки версий (такая как создание README-файла с соответствующим содержанием) может сделать исходный код настолько же воспринимаемым, насколько воспринимаема сама конструкция.

Если требуется действенное напоминание важности данных качеств, то следует помнить о том, что здравомыслие, с которым пишутся прозрачные и воспринимаемые системы, вполне может гарантировать спокойствие в будущем.

6.1. Учебные примеры

Обычной практикой в этой книге было чередование учебных примеров с философией. Данная глава начинается с рассмотрения нескольких примеров Unix-конструкций, которые демонстрируют прозрачность и воспринимаемость, а попытка извлечения из них уроков сделана после представления всех примеров. Каждый важный момент анализа во второй половине главы формулирует несколько таких уроков, а их расположение предотвращает ссылки на последующие учебные примеры, которые еще не были рассмотрены читателями.

6.1.1. Учебный пример:

audacity

Прежде всего, рассмотрим пример прозрачности в конструкции пользовательского интерфейса. Программа с открытым исходным кодом

audacity представляет собой редактор звуковых файлов, работающий в операционных системах Unix, Mac OS X и Windows. Исходные коды, загружаемые бинарные файлы, документация и снимки экранов доступны на сайте проекта <http://audacity.sourceforge.net/>.

Данная программа поддерживает операцию вырезания и вставки, а также редактирования аудио-выборок. В ней поддерживается редактирование нескольких дорожек и микширование. Пользовательский интерфейс чрезвычайно прост. Звуковые колебания отображаются в окне

audacity . Изображение звуковой волны можно редактировать с помощью вырезания и вставки; результаты операций непосредственно отражаются на аудио-выборке по мере их осуществления.

Многодорожечное редактирование поддерживается простейшим способом. Экран разделяется на несколько дисплеев (по одному для каждой дорожки), расположенных в пространстве окна таким образом, чтобы передать совпадение дорожек по времени и облегчить подбор функций путем визуального контроля. Дорожки можно перемещать с помощью мыши вправо или влево для изменения их относительной синхронизации.

Несколько функций пользовательского интерфейса реализованы превосходно и достойны подражания: крупные, легко различимые и удобные функциональные кнопки с характерными цветами, возможность отмены операции, устраняющая риск экспериментов, регулятор громкости, который своей формой визуально указывает громкость звука.

Рис. 6.1. Копия экрана программы audacity

Кроме указанных деталей, главным достоинством программы является то, что она имеет весьма прозрачный и естественный пользовательский интерфейс, который создает как можно меньше препятствий между пользователем и звуковым файлом.

6.1.2. Учебный пример: параметр -v программы

fetchmail

fetchmail — программа-шлюз. Ее главной задачей является преобразование между протоколами удаленной загрузки почты POP3 или IMAP и собственным протоколом Internet SMTP для обмена почтой. Он чрезвычайно широко распространен на Unix-машинах, использующих непостоянные SLIP- или PPP-подключения к Internet-провайдерам, и по существу, вероятно, охватывает заметную долю почтового трафика в Internet.

В

fetchmail имеется не менее 60 параметров командной строки (возможно, как будет установлено далее в данной книге, это слишком много) и большое количество других параметров, устанавливаемых не из командной строки, а из конфигурационного файла. Среди этих параметров важнейшим является -v, параметр отображения подробной информации.

При использовании параметра -v программа

fetchmail отправляет на стандартный вывод распечатки POP-, IMAP- и SMTP-транзакций по мере их совершения. Разработчик в режиме реального времени может фактически увидеть код выполнения протокола с удаленными почтовыми серверами и программой транспортировки почты. Пользователи могут отправлять распечатки сеансов с отчетами об ошибках. Ниже приведен пример характерной распечатки сеанса (см. пример 6.1).Пример 6.1. Распечатка

fetchmail -v

fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP)

at Mon, 09 Dec 2002 08:41:37 -0500 (EST): poll started

fetchmail: running ssh %h /usr/sbin/imapd

(host hurkle.thyrsus.com service imap)

fetchmail: IMAP< * PREAUTH [42.42.1.0] IMAP4rev1 v12.264 server ready

fetchmail: IMAP> A0001 CAPABILITY

fetchmail: IMAP< * CAPABILITY IMAP4 IMAP4REV1 NAMESPACE IDLE SCAN

SORT MAILBOX-REFERRALS LOGIN-REFERRALS AUTH=LOGIN

THREAD=ORDEREDSUBJECT

fetchmail: IMAP< A0001 OK CAPABILITY completed

fetchmail: IMAP> A0002 SELECT "INBOX"

fetchmail: IMAP< * 2 EXISTS

fetchmail: IMAP< * 1 RECENT

fetchmail: IMAP< * OK [UIDVALIDITY 1039260713] UID validity status

fetchmail: IMAP< * OK [UIDNEXT 23982] Predicted next UID

fetchmail: IMAP< * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)

fetchmail: IMAP< * OK [PERMANENTFLAGS

(* \Answered \Flagged \Deleted \Draft \Seen)]

Permanent flags

fetchmail: IMAP< * OK [UNSEEN 2] first unseen in /var/spool/mail/esr

fetchmail: IMAP< A0002 OK [READ-WRITE] SELECT completed

fetchmail: IMAP> A0003 EXPUNGE

fetchmail: IMAP< A0003 OK Mailbox checkpointed, no messages expunged

fetchmail: IMAP> A0004 SEARCH UNSEEN

fetchmail: IMAP< * SEARCH 2

fetchmail: IMAP< A0004 OK SEARCH completed

2 messages (1 seen) for esr at hurkle.thyrsus.com.

fetchmail: IMAP> A0005 FETCH 1:2 RFC822.SIZE

fetchmail: IMAP< * 1 FETCH (RFC822.SIZE 2545)

fetchmail: IMAP< * 2 FETCH (RFC822.SIZE 8328)

fetchmail: IMAP< A0005 OK FETCH completed

skipping message esr@hurkle.thyrsus.com:1 (2545 octets) not flushed

fetchmail: IMAP> A0006 FETCH 2 RFC822.HEADER

fetchmail: IMAP< * 2 FETCH (RFC822.HEADER {1586}

reading message esr@hurkle.thyrsus.com:2 of 2 (1586 header octets)

fetchmail: SMTP< 220 snark.thyrsus.com ESMTP Sendmail 8.12.5/8.12.5;

Mon, 9 Dec 2002 08:41:41 -0500

fetchmail: SMTP> EHLO localhost

fetchmail: SMTP< 250-snark.thyrsus.com

Hello localhost [127.0.0.1], pleased to meet you

fetchmail: SMTP< 250-ENHANCEDSTATUSCODES

fetchmail: SMTP< 250-8BITMIME

fetchmail: SMTP< 250-SIZE

fetchmail: SMTP> MAIL FROM:<mutt-dev-owner@mutt.org> SIZE=8328

fetchmail: SMTP< 250 2.1.0 <mutt-dev-owner@mutt.org>... Sender ok

fetchmail: SMTP> RCPT TO:<esr@localhost>

fetchmail: SMTP< 250 2.1.5 <esr@localhost>... Recipient ok

fetchmail: SMTP> DATA

fetchmail: SMTP< 354 Enter mail, end with "." on a line by itself

#

fetchmail: IMAP<)

fetchmail: IMAP< A0006 OK FETCH completed

fetchmail: IMAP> A0007 FETCH 2 BODY.PEEK[TEXT]

fetchmail: IMAP< * 2 FETCH (BODY[TEXT] {6742}

(6742 октета тела сообщения)

**********.**********.

**********.**********.*******

*********.

**********.**********

fetchmail: IMAP<)

fetchmail: IMAP< A0007 OK FETCH completed

fetchmail: SMTP>. (EOM)

fetchmail: SMTP< 250 2.0.0 gB9ffWo08245 Message accepted for delivery

flushed

fetchmail: IMAP> A0008 STORE 2 +FLAGS (\Seen \Deleted)

fetchmail: IMAP< * 2 FETCH (FLAGS (\Recent \Seen \Deleted))

fetchmail: IMAP< A0008 OK STORE completed

fetchmail: IMAP> A0009 EXPUNGE

fetchmail: IMAP< * 2 EXPUNGE

fetchmail: IMAP< * 1 EXISTS

fetchmail: IMAP< * 0 RECENT

fetchmail: IMAP< A0009 OK Expunged 1 messages

fetchmail: IMAP> A0010 LOGOUT

fetchmail: IMAP< * BYE hurkle IMAP4rev1 server terminating connection

fetchmail: IMAP< A0010 OK LOGOUT completed

fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP)

at Mon, 09 Dec 2002 08:41:42 -0500: poll completed

fetchmail: SMTP> QUIT

fetchmail: SMTP< 221 2.0.0 snark.thyrsus.com closing connection

fetchmail: normal termination, status 0

Параметр -v делает программу

fetchmail воспринимаемой (предоставляя возможность просмотреть обмен данными протокола). Это

чрезвычайно полезно. Я посчитал это настолько важным, что написал специальный код для маскирования паролей учетных записей в распечатках транзакций, выполненных благодаря параметру -v, так чтобы распечатки можно было отправлять без необходимости редактирования секретной информации в них.

Это оказалось хорошим сигналом. По крайней мере восемь из десяти проблем, представленных в отчетах, диагностировались хорошо осведомленными специалистами в течение нескольких секунд при просмотре распечаток сеансов. В списке рассылки fetchmail числится несколько знающих людей. По существу, ввиду того, что большинство программных сбоев легко диагностируются, автору редко приходилось разбираться с ними самостоятельно.

Со временем

fetchmail приобрела репутацию "пуленепробиваемой" программы. Ее можно неправильно настроить, однако полные отказы происходят очень редко. Это ничто по сравнению с возможностью быстро получить точную информацию по поводу восьми из десяти ошибок.

Из данного примера можно извлечь следующий урок: не следует с опозданием реализовывать отладочные инструменты или рассматривать их как одноразовое средство. Они являются окнами в код. Не достаточно просто "пробивать грубые отверстия в стенах", их необходимо "отделывать и остеклять". Если код должен быть сопровождаемым, то всегда приходится "пускать в него свет".

6.1.3. Учебный пример: GCC

Программа GCC, GNU C-компилятор, применяемый в большинстве современных Unix-систем, возможно, наилучшим образом демонстрирует преимущества проектирования с учетом прозрачности. Программа GCC организована как последовательность стадий обработки, связанных вместе программой драйвера: стадии препроцессора, синтаксического анализатора, генератора кода, ассемблера и линкера.

На каждой из первых трех стадий принимается и генерируется читабельный текстовый формат (ассемблер должен создавать, а линкер принимать двоичные форматы, почти по

определению). С помощью различных параметров командной строки для драйвера

gcc(1) можно получить не только результаты после обработки С-кода препроцессором, сборки и создания объектного кода, но и отслеживать результаты множества промежуточных этапов синтаксического анализа и генерации кода.

Это в точности структура cc, первого (PDP-11) С-компилятора. Кен Томпсон.

Существует множество преимуществ такой организации. Одним из них и особенно важным для GCC является возвратное тестирование[59]. Поскольку большинство различных промежуточных форматов являются текстовыми, отклонения от ожидаемых результатов в возвратном тестировании легко предсказываются и анализируются с помощью простых текстовых diff-операций над промежуточными результатами. Нет необходимости использовать специальные средства дамп-анализа, которые, вполне возможно, имеют собственные ошибки, и в любом случае будут вносить дополнительные сложности при обслуживании.

Данный пример позволяет сформулировать модель проектирования, которая заключается в том, что программа драйвера имеет мониторинговые ключи, которые просто (но в достаточной степени) отображают потоки текстовых данных между компонентами. Как и в случае с параметром -v программы

fetchmail, подобная возможность не является запоздалой доработкой, она встроена в конструкцию, чтобы улучшить ее воспринимаемость.

6.1.4 Учебный пример:

kmail

kmail — программа с графическим пользовательским интерфейсом для чтения почтовых сообщений, распространяемая в составе среды KDE. Пользовательский интерфейс разработан со вкусом, хорошо спроектирован и имеет множество полезных функций, включая автоматическое отображение вложенных изображений в MIME-вложениях и поддержку шифрования/дешифрования PGP-ключей. GUI-интерфейс программы дружественный по отношению к конечным пользователям, включая нетехнических.

Во многих пользовательских почтовых агентах разработчики делают один шаг в сторону воспринимаемости, встраивая команду, позволяющую переключать режим отображения всех почтовых заголовков в противоположность отображению только нескольких, например "From" и "Subject". Пользовательский интерфейс

kmail гораздо дальше продвинулся в этом направлении.

Работающая программа

kmail отображает уведомления о состоянии в однострочном субокне в нижней части основного окна, иными словами, в компактной строке состояния, явно смоделированной с аналогичного элемента Netscape/Mozilla. Когда пользователь открывает почтовый ящик, в строке состояния отображается общее количество сообщений и количество непрочитанных сообщений. Проигнорировать уведомления легко, а в случае необходимости можно также без труда уделить им внимание.

Графический интерфейс

kmail является примером хорошей конструкции пользовательского интерфейса. Он информативен, но не отвлекает внимание. Он организован вокруг идеи о том, что лучшей политикой для нормально функционирующих инструментов Unix является тишина (см. гл. 11). Авторы продемонстрировали превосходный вкус, заимствовав вид и восприятие строки состояния браузера.

Однако мера вкуса разработчиков программы становится ясна только при необходимости поиска неисправностей установленной системы, в которой возникают проблемы при отправке почты. Если внимательно наблюдать за процессом отправки, то будет заметно, что каждая строка SMTP-транзакции с удаленным почтовым сервером по мере выполнения отображается в строке состояния

kmail.

Разработчики

kmail умело избегают ловушки, которая часто делает GUI-программы, подобные

kmail, источником серьезных проблем для специалистов по устранению неисправностей. Большинство коллективов разработчиков, преследующих аналогичные kmail цели, полностью избавлялись бы от таких сообщений, опасаясь, что они подтолкнут нетехнических пользователей к возвращению к показной псевдопростоте Windows.

Вместо этого разработчики

kmail проектировали прозрачную программу. Они сделали сообщения транзакций видимыми, но также создали простую возможность визуально их игнорировать. Верно выбрав форму представления, они сумели удовлетворить требования как нетехнических пользователей, так и опытных специалистов Unix. Это было блестящим решением. Данной методике можно и нужно подражать, разрабатывая другие GUI-интерфейсы.

Рис. 6.2. Копия экрана kmail

Видимость данных сообщений, несомненно, полезна для неискушенного пользователя. Данные сообщения помогают и специалистам, которые пытаются решить проблемы с почтой, возникшие у этого пользователя.

Урок в данном случае очевиден. Заставить пользовательский интерфейс "молчать" — только наполовину изящное решение. Действительно изящным будет найти способ оставить подробности доступными, но сделать их ненавязчивыми.

6.1.5. Учебный пример: SNG

Программа

sng осуществляет преобразование формата PNG в его полнотекстовое представление (формат SNG или Scriptable Network Graphics) и обратно. Формат SNG можно просматривать и модифицировать с помощью обычного текстового редактора. Работающая с PNG-файлом программа создает SNG-файл, а при запуске для SNG-файла она восстанавливает эквивалентный PNG-файл. Преобразование осуществляется исключительно точно, без потерь и в обоих направлениях.

Стиль синтаксиса SNG подобен CSS (Cascading Style Sheets — каскадные таблицы стилей),

другому языку для управления представлением графики, что делает, как минимум, шаг в сторону правила наименьшей неожиданности. Ниже приводится тестовый пример. Пример 6.2. SNG-файл

```
#SNG: This is a synthetic SNG test file

#(Искусственный тестовый SNG-файл)


# Our first test is a paletted (type 3) image.

#(Первый тест - индексированное (тип 3) изображение.)

IHDR: {

 width: 16;

 height: 19;

 bitdepth: 8;

 using color: palette;

 with interlace;

}


#

# Sample bit depth chunk (Блок глубины цвета)

sBIT: {

 red: 8;

 green: 8;

 blue: 8;

}


# An example palette: three colors, one of which

# we will render transparent

#(пример палитры: три цвета, один из #которых выводится прозрачным)

PLTE: {

 (0, 0, 255)

 (255, 0, 0)

 "dark slate gray",
```

```
}
# Suggested palette (Рекомендованная палитра)
sPLT {
 name: "A random suggested palette";
 depth: 8;
 (0, 0, 255), 255, 7;
 (255, 0, 0), 255, 5;
 ( 70, 70, 70), 255, 3;
}

# The viewer will actually use this
#(программа просмотра фактически
#использует такие данные)...
IMAGE: {
 pixels base64
2222222222222222
2222222222222222
0000001111100000
0000011111110000
0000111001111000
0001110000111100
0001110000111100
0000110001111000
0000000011110000
0000000111100000
0000001111000000
0000001111000000
0000000000000000
0000000110000000
0000001111000000
```

```
0000001111000000

0000000110000000

2222222222222222

2222222222222222

}


tEXt: {

 # Ordinary text chunk (Обычный текстовый блок)

 keyword: "Title";

 text: "Sample SNG script";

}


# Test file ends here

#(Окончание тестового файла)
```

Цель данного инструментального средства — позволить пользователю редактировать различные непонятные типы PNG-блоков, которые вовсе не обязательно поддерживаются традиционными графическими редакторами. Вместо написания специализированного кода для анализа двоичного PNG-формата, пользователь может просто преобразовать изображение в полнотекстовое представление, отредактировать его, а затем переконвертировать его обратно. Другая потенциальная прикладная задача заключается в том, чтобы сделать изображение открытым для систем контроля версий. В большинстве систем контроля версий текстовыми файлами гораздо проще управлять, чем большими двоичными блоками, а diff-операции над SNG-файлами фактически имеют некоторые возможности для извлечения полезных сведений.

Однако выигрыш в данном случае связан не только со временем, не потраченным на написание специализированного кода для манипуляций с двоичными PNG-файлами. Код программы

sng не является очень прозрачным, однако он поддерживает прозрачность в более крупных системах программ, делая все содержимое PNG-файлов воспринимаемым.

6.1.6. Учебный пример: база данных Terminfo

База данных terminfo представляет собой набор описаний видеотерминалов. В каждой записи описываются escape-последовательности, которые осуществляют различные операции на экране терминала, такие как вставка или удаление строк, удаление символов от курсора до конца строки или экрана, или начало и завершение подсветки экрана, такой как негативное видеоизображение, подчеркивание или мерцание.

База данных terminfo главным образом используется библиотеками

curses(3) , которые лежат в основе rogue-подобного стиля интерфейса, описанного в главе 11, и некоторых широко используемых программ, таких как

mutt(1), lynx(1) и

slrn(1) . Хотя эмуляторы терминалов, такие как

xterm(1), работающие на современных растровых дисплеях, обладают всеми возможностями, которые незначительно отличаются от возможностей стандарта ANSI X3.64 и устаревших терминалов VT100, существует достаточно много таких разновидностей терминалов, где жесткая привязка приложения к ANSI-возможностям была бы плохой идеей. База данных terminfo также достойна рассмотрения, ввиду того, что проблемы, логически подобные проблемам, которые решаются terminfo, постоянно возникают в управлении другими видами периферийного аппаратного обеспечения, не имеющего стандартного способа предоставления информации о собственных характеристиках.

В конструкции terminfo учтен опыт более раннего формата описания характеристик, который называется termcap. База данных termcap-описаний в текстовом формате содержалась в одном большом файле, /etc/termcap. Несмотря на то, что данный формат в настоящее время является устаревшим, почти в каждой Unix-системе имеется его копия.

Обычно ключом, используемым для поиска записи о типе терминала, является переменная среды TERM, способ установки которой для целей данного учебного примера не имеет особого значения[60]. Приложения, использующие terminfo (или termcap), вносят небольшую задержку при запуске. Когда библиотека

curses(3) инициализируется, она должна найти запись, соответствующую значению переменной TERM, и загрузить эту запись в память.

Практика работы с termcap показала, что на задержку при запуске доминирующее влияние оказывало время, необходимое для синтаксического анализа текстовой формы представления характеристик. Поэтому terminfo-записи являются копиями двоичных структур, для которых операции маршалинга и демаршалинга выполняются быстрее. Существует главный текстовый формат для всей базы данных, файл характеристик terminfo. Данный файл (или отдельные записи) можно преобразовать в двоичную форму с помощью terminfo-компилятора

tic(1) , а бинарные записи декомпилируются в редактируемый текстовый формат посредством утилиты

infocmp(1) .

Такая конструкция внешне противоречит данным в главе 5 рекомендациям против использования двоичного кэширования, однако, в сущности, это тот экстремальный случай, в котором данный подход является хорошей тактикой. Главные текстовые файлы редактируются очень редко — в действительности Unix-системы обычно поставляются с предварительно скомпилированной базой данных terminfo, а главный текстовый файл служит в основном в качестве документации. Следовательно, проблемы синхронизации и несогласованности, которые обычно препятствуют реализации такого подхода, почти никогда не возникают.

Проектировщики terminfo могли бы оптимизировать скорость другим путем. Вся база данных бинарных записей могла бы размещаться в некотором большом трудном для понимания файле базы данных. Однако они выбрали более мудрое решение, которое к тому же больше

соответствовало бы духу операционной системы Unix. Записи terminfo располагаются в иерархии каталогов, обычно в современных Unix-системах в каталоге /usr/share/terminfo. Точное расположение базы данных в конкретной системе можно выяснить с помощью страницы документации

terminfo(5) .

Изучая каталог terminfo, можно заметить, что в качестве имен подкаталогов используются одиночные печатаемые символы. В каждом подкаталоге находятся записи для всех типов терминалов, имена которых начинаются с данной буквы. Цель такой организации заключалась в том, чтобы избежать необходимости выполнять линейный поиск в очень большом каталоге. В более современных файловых системах Unix, которые представляют каталоги с помощью бинарных деревьев или других структур, оптимизированных для быстрого поиска, подкаталоги не являются необходимыми.

Я выяснил, что даже в довольно современных Unix-системах разделение большого каталога на подкаталоги может существенно повысить производительность. На поздней модели DEC Alpha, использующей DEC Unix, были десятки тысяч файлов базы данных авторизованных пользователей для крупного образовательного учреждения. (Из всех протестированных схем наилучшим образом функционировали подкаталоги с именами, состоящими из первой и последней букв фамилии, например, записи для "johnson" находились бы в каталоге "j_n". Использование первых двух букв было не настолько эффективным, поскольку было множество систематически создаваемых имен, которые отличались только окончанием.) Это, возможно, говорит только о том, что сложное индексирование каталогов до сих пор является не таким распространенным, каким должно было бы быть... Однако даже в этом случае оно делает организацию, хорошо работающую без него, более переносимой, чем та, которая в нем нуждается. Генри Спенсер.

Таким образом, издержки открытия terminfo-записи сводятся к двум операциям поиска в файловой системе и открытию файла. Но поскольку извлечение той же записи из одной большой базы данных потребовало бы поиска и открытия базы данных, инкрементальные издержки организации terminfo равны максимум одной операции поиска в файловой системе. Фактически они меньше. Существует разница затрат между одной операцией поиска в файловой системе и каким бы то ни было методом поиска, применяемым в одной большой базе данных. Однократное использование такого метода при запуске приложения, вероятно, не влечет серьезных последствий и вполне допустимо.

В качестве иерархической базы данных terminfo использует саму файловую систему. Это превосходный пример "творческой лени", согласованной с правилами экономии и прозрачности. Это означает, что все обычные инструментальные средства для навигации, изучения и модификации файловой системы могут применяться для навигации, изучения и модификации базы данных terminfo. Не требуется создавать и отлаживать какие-либо специальные средства (отличные от

tic(1) и

infocmp(1) для упаковки и распаковки отдельных записей). Это также означает, что работа над ускорением доступа к базе данных была бы работой по ускорению самой файловой системы, что создало бы преимущества для гораздо большего количества приложений, а не только для программ, использующих библиотеки

curses(3) .

Существует одно дополнительное преимущество такой организации, которое не характерно для случая с terminfo. Имеется возможность использовать Unix-механизм привилегий вместо необходимости создавать собственный уровень управления доступом с его собственными

ошибками. Это является следствием принятия философии Unix "все является файлом", а не попыткой противостояния ей.

Расположение каталога terminfo является весьма неэффективным использованием пространства на большинстве файловых систем Unix. Длина записей обычно находится в диапазоне от 400 до 1400 байт, а файловые системы обычно выделяют минимум 4 Кб для каждого непустого файла на диске. Разработчики пошли на такие издержки по той же причине, по которой выбрали упакованный двоичный формат, т.е. для того, чтобы свести к минимуму задержку при запуске в программах, использующих terminfo. С тех пор емкость диска при постоянной цене возросла в тысячи раз, оправдывая данное решение.

Поучителен контраст с форматами, применяемыми в файлах реестра Microsoft Windows. Реестры представляют собой базы данных свойств, используемые как самой операционной системой Windows, так и прикладными программами в ней. Каждый реестр располагается в одном большом файле. Реестры содержат смесь текстовых и бинарных данных, для редактирования которой требуются специализированные инструментальные средства. Среди прочих недостатков подход одного большого файла приводит к печально известному феномену "сползания реестра". Среднее время доступа безгранично возрастает, по мере того как добавляются новые записи. Поскольку стандартного, предоставленного системой API-интерфейса для редактирования реестра не существует, приложения используют специализированный код для самостоятельного редактирования, создавая печально известную опасность повреждения, которое может заблокировать всю систему.

Использование файловой системы Unix в качестве базы данных является хорошей тактикой для подражания в других приложениях с простыми требованиями к базе данных. Весомые причины не делать этого, вероятнее всего, связаны с необходимостью обрабатывать ключи базы данных, которые не выглядят как естественные имена файлов, а не с какими-либо проблемами производительности. В любом случае, это хороший и быстрый прием, который может быть очень полезен при создании прототипов.

6.1.7. Учебный пример: файлы данных Freeciv

Игра Freeciv — стратегия с открытым исходным кодом, прообразом которой послужила классическая игра

Civilization II Сида Мейера (Sid Meier). Игра состоит в том, что каждый участник, имея в своем распоряжении группу кочевников каменного века, строит свою цивилизацию. Происходит борьба этих цивилизаций, созданных игроками, в ходе исследования и колонизации мира, военных сражений, развития торговых отношений и технологических новшеств. В качестве игрока может выступать и искусственный интеллект. Выиграть можно, либо "завоевав мир", либо первым достигнув технологического уровня, достаточного для отправки космического корабля к Альфа Центавра. Исходные коды и документация доступны на сайте проекта <http://www.freeciv.org/>.

В главе 7 стратегическая игра Freeciv рассматривается как пример клиент-серверного разделения, в котором сервер поддерживает общее состояние, а клиент концентрируется на .GUI-представлении. Однако данная игра также имеет другую примечательную архитектурную особенность. Большая часть фиксированных данных игры вместо того, чтобы быть встроенной в код сервера, выражается в реестре свойств, который считывается игровым сервером во время запуска игры.

Файлы реестра игры записываются в текстовом формате, в котором текстовые строки (со

связанным текстом и числовыми свойствами) группируются в различные внутренние списки важных данных (таких как нации и типы активных единиц) на игровом сервере. Мини-язык имеет директиву include, и данные игры могут быть разбиты на семантические блоки (различные файлы), каждый из которых можно редактировать по отдельности. Такой выбор конструкции поддерживается до такой степени, что можно определять новые нации и новые типы активных элементов просто путем создания новых определений в файлах данных, абсолютно не затрагивая код сервера.

Синтаксический анализ при запуске сервера Freeciv обладает одной интересной особенностью, которая создает некоторый конфликт между двумя правилами Unix-проектирования и поэтому достойна более подробного рассмотрения. Сервер игнорирует имена свойств, если не имеет информации о том, как использовать данные свойства. Это делает возможным объявление свойств, которые сервер еще не использует, без изменения начального синтаксического анализа. Это означает, что разработка данных игры (политики) и серверного ядра (механизма) могут быть полностью обособлены. С другой стороны, это также означает, что в ходе синтаксического анализа при запуске не будут обрабатываться простые синтаксические ошибки в именах атрибутов. Этот скрытый сбой, по-видимому, нарушает правило исправности.

Для того чтобы разрешить данное противоречие, необходимо отметить, что

использование данных реестра входит в задачи сервера, но задача тщательной проверки ошибок в этих данных может быть передана другой программе, запускаемой человеком, редактирующим реестр при каждом его изменении. Одним из решений в духе Unix было бы использование отдельной программы ревизии, которая анализирует либо машинно-считываемую спецификацию формата правил, либо источник серверного кода для определения набора используемых им свойств, выполняет синтаксический анализ реестра Freeciv для определения набора предоставляемых им свойств и формирует отчет об отличиях[61].

Совокупность всех файлов данных функционально подобна реестру Windows и даже использует синтаксис, аналогичный текстовым частям реестров. Однако проблемы сползания и повреждения в данном случае не возникают, поскольку ни одна программа (внутри или вне набора Freeciv)

не записывает информацию в эти файлы. Реестр Freeciv доступен только для чтения и редактируется только кураторами игры.

Влияние синтаксического анализа файлов данных на производительность сведено к минимуму, так как для каждого файла данная операция производится только один раз во время запуска клиента либо сервера.

Рис. 6.3. Главное окно игры Freeciv

6.2. Проектирование, обеспечивающее прозрачность и воспринимаемость

Для того чтобы проектировать программы с учетом прозрачности и воспринимаемости, необходимо применять все тактические приемы для сохранения простоты кода, а также уделять особое внимание способам, которые облегчат обмен информацией между разработчиками. После решения вопроса о том, будет ли данная конструкция работать, первое, что необходимо выяснить, — будет ли код удобочитаемым для других разработчиков и является ли он изящным? Авторы выражают надежду, что к настоящему моменту читателю

ясно, что данные вопросы не являются формальными и что изящество кода — не роскошь. Это очень важно для сокращения количества ошибок и увеличения возможности долгосрочного сопровождения поддержки.

6.2.1. Дзэн прозрачности

Проявляющаяся в рассмотренных выше примерах модель заключается в следующем: наиболее эффективный способ создания прозрачного кода заключается в том, чтобы просто не создавать чрезмерное количество уровней абстракции над той реальной проблемой, которую будет решать данный код.

В разделе главы 4 о значении освобождения было рекомендовано абстрагироваться, упрощать и обобщать, "очищаться" и "освобождаться" от частных, случайных условий, при которых была сформулирована проектная задача. Совет абстрагироваться, в сущности, не противоречит сформулированной здесь рекомендации не создавать излишней абстракции, поскольку между освобождением от предположений и потерей из вида решаемой проблемы имеется существенное отличие. Это непосредственно связано с идеей о необходимости сохранять тонкие связующие уровни.

Один из уроков Дзэн заключается в том, что мы обычно видим мир "сквозь пелену" предубеждений и застывших идей, которые являются продолжением наших желаний. Чтобы достичь просветления, нужно следовать учению Дзэн не только для освобождения от желаний и привязанностей, а для того, чтобы осознать реальность в точности такой, какова она есть, т.е. без зацикливания на предубеждениях и навязчивых идеях.

Это превосходный прагматичный совет для разработчиков программного обеспечения. Он напрямую связан с тем, что подразумевается в классическом для Unix совете быть минималистом. Разработчики программного обеспечения — талантливые люди, формирующие идеи (абстракции), касающиеся прикладных областей, которыми они занимаются. Вокруг этих идей они организовывают создаваемые программы, а затем при отладке часто обнаруживают, что создали для себя серьезные проблемы, рассматривая происходящее сквозь призму собственных идей.

Любой учитель Дзэн немедленно распознал бы данную проблему и, возможно, наказал бы ученика. Осознанное проектирование с учетом прозрачности является несколько "менее мистическим" способом решения данной проблемы.

В главе 4 дан критический обзор объектно-ориентированного программирования, который, вероятно, несколько шокировал программистов, выросших на ОО-учении 90-х годов прошлого века. Конструкции, основанные на объектно-ориентированных языках программирования, не должны быть чрезмерно сложными, но, как отмечалось в главе 4, часто являются таковыми. Слишком многие ОО-конструкции являются хитросплетениями отношений "является" и "содержит" (is-a и has-a) или характеризуются большими связующими уровнями, в которых многие объекты, кажется, существуют только для того, чтобы занимать место в неприступной пирамиде абстракций. Подобные конструкции противоположны прозрачным, они печально известны как неясные и сложные для отладки.

Как отмечалось выше, Unix-программисты с самого начала являются приверженцами модульности, однако стремятся реализовать ее более незаметным способом. Сохранение тонких связующих уровней — часть такого подхода. В более общем смысле традиции учат нас создавать более низкие модули, связанные с основой с помощью алгоритмов и структур, которые спроектированы как простые и прозрачные.

Как и в случае искусства Дзэн, простота хорошего кода в Unix зависит от строгой самодисциплины и высокого уровня мастерства, которые не обязательно видны при случайном рассмотрении. Создание прозрачности — тяжелый труд, но он стоит усилий не просто из соображений искусства. В отличие от Дзэн-искусства, программное обеспечение требует отладки и обычно нуждается в продолжительном сопровождении, дальнейшем переносе на другие платформы и адаптации в течение жизненного цикла. Следовательно, прозрачность — это более чем эстетический триумф. Прозрачность — победа, которая отражается в более низких затратах в течение жизненного цикла программного обеспечения.

6.2.2. Программирование, обеспечивающее прозрачность и воспринимаемость

Прозрачность и воспринимаемость, подобно модульности, в основном являются свойствами конструкции, а не кода. Не достаточно получить правильные низкоуровневые элементы стиля, такие создание отступов в коде ясным и последовательным способом или использование хороших соглашений по именованию переменных. Данные качества гораздо сильнее связаны со свойствами кода, которые менее очевидны при просмотре. Ниже приводятся некоторые из них.

• Какова максимальная глубина иерархии вызова процедур? Т.е., не считая рекурсии, сколько уровней вызова человеку придется мысленно смоделировать, для того чтобы понять работу кода? Совет: будьте предельно внимательны, та как наличие более четырех уровней явно свидетельствует о возникшей проблеме.

• Имеются ли в коде инвариантные свойства[62], строгие и видимые одновременно? Инвариантные свойства помогают человеку анализировать код и обнаруживать проблемные случаи.

• Являются ли вызовы функций в API-интерфейсах индивидуально ортогональными, или имеют ли они чрезмерное количество "магических" флагов и битов режима, которые заставляют один вызов выполнять несколько задач? Полный отказ от флагов режима может привести к созданию загроможденного API-интерфейса с чрезмерным количеством почти идентичных функций. Но еще шире распространена противоположная ошибка (множество флагов режима, которые легко забыть или перепутать).

• Существует ли несколько выступающих структур данных или одна глобальная таблица, собирающая высокоуровневые данные о состоянии системы? Просто ли визуализировать и проверить данное состояние, или оно рассеяно среди множества индивидуальных глобальных переменных или объектов, которые трудно найти?

• Существует ли в программе четкое, точное соответствие между структурами данных или классами и объектами реального мира, которые представлены ими?

• Просто ли отыскать часть кода, ответственную за любую заданную функцию? Как много внимания было уделено читабельности не только отдельных функций и модулей, но и кода в целом?

• Создает ли код частные случаи или избегает их? Каждый частный случай мог бы взаимодействовать со всеми остальными частными случаями, и все эти потенциальные коллизии являются ошибками, ожидающими своего часа. Но еще более важно то, что частные случаи усложняют понимание кода.

• Каково количество "магических" чисел (необъяснимых констант) в коде? Просто ли

обнаружить ограничения реализации (такие как критические размеры буферов) путем просмотра?

Лучше всего, если код будет простым. Однако если проверка кода дает хорошие ответы на приведенные выше вопросы, то код может быть весьма сложным и вместе с тем не создавать невыполнимой когнитивной нагрузки на кураторов.

Читатели могут найти полезным сравнение данных вопросов с контрольным списком вопросов, касающихся модульности в главе 4.

6.2.3. Прозрачность и предотвращение избыточной защищенности

Близким родственником присутствующей в среде программистов тенденции создавать чрезмерно сложные нагромождения абстракций является стремление чрезмерно оберегать остальных от низкоуровневых деталей. Несмотря на то, что скрывать такие детали в обычном режиме работы программы не является плохой практикой (например, в программе

fetchmail ключ -v по умолчанию не используется), низкоуровневые подробности должны легко обнаруживаться. Имеется важное различие между их сокрытием и недоступностью.

Программы,

не способные показать, что они выполняют, значительно усложняют поиск и устранение неисправностей. Поэтому опытные пользователи операционной системы Unix действительно воспринимают наличие отладочных ключей и оснащенность средствами контроля как хороший знак, а их отсутствие — как плохой. Отсутствие говорит о неопытности или небрежности разработчика. В то же время их наличие означает, что разработчик достаточно предусмотрителен, чтобы придерживаться правила прозрачности.

Соблазн чрезмерно скрывать детали особенно сильно проявляется в предназначенных для конечных пользователей GUI-приложениях, таких как программы чтения почты. Одной из причин, по которой Unix-разработчики прохладно воспринимают GUI-интерфейсы, является то, что поспешность проектировщиков таких интерфейсов в стремлении сделать их "дружественными к пользователю" часто делает их безнадежно закрытыми для любого, кто вынужден решать проблемы пользователей или должен взаимодействовать с интерфейсом за пределами узкого диапазона, предсказанного разработчиком пользовательского интерфейса.

Еще хуже то, что программы, которые скрывают выполняемые операции, как правило, содержат в себе множество предположений и являются слабыми или ненадежными, или характеризуются и тем, и другим недостатком при любом использовании, не предусмотренном разработчиком. Инструменты, которые выглядят безукоризненно, но разрушаются под воздействием нагрузки, не обладают высокой долгосрочной ценностью.

Unix-традиции настаивают на создании программ, которые являются гибкими для более широкого диапазона использования и ситуаций поиска и устранения неисправностей, включая способность предоставлять пользователю столько сведений о состоянии и активности, сколько он требует. Эта особенность полезна для поиска и устранения неисправностей; она также полезна более сообразительным и уверенным в своих силах пользователям.

6.2.4. Прозрачность и редактируемые формы представления

Другой идеей, возникающей в связи с данными примерами, является значение программ, которые переводят проблему из области, где обеспечить прозрачность трудно, в область, где реализовать ее легко. Программы audacity,

sng(1) и пара

tic(1)/infocmp(1) обладают данным свойством. Объекты, которыми они манипулируют, "не удобны для зрения и рук". Аудиофайлы не являются визуальными объектами, а редактировать блоки PNG-аннотаций сложно, несмотря на то, что PNG-изображения видимы. Все три приложения превращают изменение своих двоичных файловых форматов в проблему, решить которую пользователи вполне могут благодаря своей интуиции и навыкам, полученным из повседневного опыта.

Правило, которому следуют все описанные в данной главе примеры, заключается в том, что они как можно меньше нарушают представление данных, а в действительности они осуществляют обратимое преобразование без потерь. Данное свойство весьма важно, и его стоит реализовывать, даже если к приложению не предъявляются очевидные требования о стопроцентной точности воспроизведения. Оно дает пользователям уверенность в том, что они могут экспериментировать с данными, не нарушая их целостности.

Все преимущества текстовых форматов файлов данных, рассмотренных в главе 5, также применимы к текстовым форматам, которые создаются утилитами

sng(1), infocmp(1) и им подобными. Одной важной прикладной задачей для

sng(1) является автоматическое создание PNG-аннотаций с помощью сценариев. Такие сценарии просты в написании, поскольку существует утилита

sng(1) .

Каждый раз при разработке конструкции, которая затрагивает редактирование некоторого вида сложного двоичного объекта, традиция Unix, прежде всего, заставляет разработчика выяснить, возможно ли написать средство, аналогичное

sng(1) или паре

tic(1)/infocmp(1) , которое способно без потерь выполнять преобразование данных в редактируемый текстовый формат и обратно. Устоявшегося термина для программ такого рода не существует, но в данной книге они называются

текстуализаторами (textualizers) .

Если двоичный объект создается динамически или имеет очень большие размеры, то может быть непрактично или невозможно охватить текстуализатором всю его структуру. В таком случае эквивалентная задача состоит в написании браузера. Принципиальным примером является утилита

fsdb(1) , отладчик файловой системы, поддерживаемый в различных Unix-системах. Существует Linux-эквивалент, который называется

debugfs(1) . Двумя другими примерами является утилита

psql(1) , используемая для просмотра баз данных PostgreSQL, и программа

smbclient(1) , которую можно применять для опроса Windows-ресурсов на Linux-машине, оснащенной пакетом SAMBA. Все эти утилиты являются простыми CLI-программами, которыми можно управлять с помощью сценариев и тестовых программ.

Написание текстуализатора или браузера является весьма полезным упражнением по крайней мере по четырем причинам.

•

Получение превосходного образовательного опыта. Могут существовать другие такие же хорошие способы для изучения структуры объектов, но нет ни одного, который был бы очевидно лучше.

•

Возможность создавать дампы содержимого структуры для просмотра и отладки. Такие инструменты упрощают создание дампов. Следовательно, появляется возможность получать больше информации, что, вероятно, ведет к более глубокому пониманию.

•

Возможность свободно создавать тестовые нагрузки и нестандартные случаи. Это означает, что появляется больше возможностей для проверки разрозненных участков пространства состояния объекта, а также возможность проверить связанное программное обеспечение и устранить проблемы еще до того, как с ними столкнутся пользователи.

•

Получение кода, который можно использовать повторно. Если тщательно подходить к написанию браузера/текстуализатора и поддерживать CLI-интерпретатор совершенно отдельно от библиотеки маршалинга/демаршалинга, то впоследствии может выясниться, что данный код можно повторно использовать для реального приложения.

После создания текстуализатора или браузера вполне может появиться возможность применить модель "разделения ядра и интерфейса" (см. главу 11), в которой созданный текстуализатор/отладчик будет использоваться как ядро. Все обычные преимущества данной модели будут также применимы.

Желательно, хотя часто это трудно сделать, чтобы текстуализатор был способен считывать и записывать даже поврежденный двоичный объект. Во-первых, это позволяет создавать контрольные примеры с повреждением данных для программ тестирования под нагрузкой. Во-вторых, это может в целом упростить экстренное восстановление. Возможно, будет трудно обработать случаи, в которых

структура объекта загрязнена, но, по крайней мере, следует обработать случаи, в которых

содержимое структуры ошибочно, например, путем преобразования бессмысленных значений в шестнадцатеричную форму и конвертирования их обратно.Генри Спенсер.

6.2.5. Прозрачность, диагностика и восстановление после сбоев

Еще одним преимуществом прозрачности, связанным с простотой отладки, является то, что в прозрачных системах проще выполнять действия по восстановлению после сбоев, и часто

такие системы, в первую очередь, более устойчивы к повреждениям от ошибок.

При сравнении базы данных terminfo с реестром операционной системы Windows отмечалось, что реестр печально известен как структура, разрушаемая приложениями с ошибочным кодом. Это может сделать недоступной всю систему. Даже если система продолжает оставаться работоспособной, могут возникнуть трудности с восстановлением, если повреждение сбивает с толку специализированные средства редактирования реестра.

Приведенные выше учебные примеры иллюстрируют способы, с помощью которых проектирование, обеспечивающее прозрачность, позволяет предотвратить проблемы данного класса. Так как база данных terminfo не содержится в одном большом файле, повреждение одной terminfo-записи не делает весь набор данных terminfo непригодным к использованию. Синтаксический анализ полностью текстовых однофайловых форматов, таких как termcap, обычно осуществляется с помощью методов, которые (в отличие от операций поблочного чтения дампов двоичной структуры) способны восстановить данные после точечных ошибок. Синтаксические ошибки в SNG-файле могут быть исправлены вручную, без необходимости использования специализированных редакторов, которые могут отказать при загрузке поврежденного PNG-изображения.

Возвращаясь к учебному примеру

kmail , можно отметить, что данная программа упрощает диагностику сбоев, поскольку подчиняется правилу исправности: информация о сбоях протокола SMTP отображается полностью, поэтому ее удобно анализировать. Нет необходимости расшифровывать множество туманных сообщений, сгенерированных самой программой

kmail , для того чтобы увидеть, как выглядит обмен данными с SMTP-сервером. Все, что требуется делать — смотреть в нужном направлении, поскольку данная программа прозрачна и не удаляет информацию об ошибках протокола. (Способствует также то, что протокол SMTP сам по себе является текстовым и включает в свои транзакции сообщения о состоянии, которые может прочесть человек.)

Средства обеспечения воспринимаемости, такие как текстуализаторы и браузеры, также упрощают диагностику сбоев. Одна из причин уже рассматривалась: они упрощают проверку состояния системы. Однако следует обратить внимание на другой эффект, связанный с их работой. Текстовые версии данных стремятся иметь полезную избыточность (например, использование пробелов для визуального разделения, а также явных разделителей для синтаксического анализа). Такая избыточность упрощает чтение данных форматов людьми, но также делает форматы более устойчивыми к безвозвратному удалению точечными сбоями. Поврежденный блок данных PNG-файла редко поддается восстановлению, однако человеческая способность распознавать модели и анализировать содержание может помочь в восстановлении эквивалентного SNG-файла.

Еще больше проясняется правило устойчивости. Простота плюс прозрачность снижает затраты, уменьшает напряжение разработчиков и освобождает людей для концентрации на новых проблемах вместо исправления старых.

6.3. Проектирование, обеспечивающее удобство сопровождения

Программное обеспечение удобно в сопровождении в той мере, в которой люди, не являющиеся его создателями, могут его понять и модифицировать. Для обеспечения удобства сопровождения требуется не просто хорошо работающий код, нужен код, который

соответствует правилу ясности и успешно взаимодействует с человеком и компьютером.

Unix-программисты обладают большим количеством доступных им скрытых знаний о том, что делает код удобным в сопровождении, поскольку Unix поддерживает исходный код, который существует десятилетиями. По причинам, которые описываются в главе 17, Unix-программисты учатся не исправлять неряшливый код, а избавляться от него и создавать более четкий код (см. размышления Роба Пайка по данной теме в главе 1). Поэтому любой исходный код, переживший более десяти лет эволюции, избран как удобный в сопровождении. Такие старые, успешные, хорошо организованные проекты с удобным в сопровождении кодом являются созданными сообществом моделями для практического применения.

"Данный код живой, замороженный или мертвый?" — это вопрос, который обычно задают Unix-программисты и особенно программисты сообщества открытого исходного кода при оценке тех или иных инструментов. Вокруг "живого" кода сосредотачивается сообщество активных разработчиков. "Замороженный" код часто становится таковым ввиду того, что создает гораздо больше проблем, чем конструктивных решений для его разработчиков. "Мертвый" код так долго пребывает в "замороженном" состоянии, что было бы проще реализовать его эквивалент с самого начала. Для того чтобы код "жил", необходимо направить максимум усилий на то, чтобы сделать код удобным в сопровождении (и, следовательно, привлекательным для будущих кураторов).

Код, разработанный как прозрачный и воспринимаемый, уже почти автоматически становится удобным в сопровождении. Однако существует и другая, не менее достойная для подражания практика, которая отображена в моделях, рассмотренных в настоящей главе.

Одним из важнейших практических принципов является применение правила ясности: использование простых алгоритмов. В главе 1 процитировано мнение Кена Томпсона: "Если вы сомневаетесь, используйте грубую силу". Томпсон понимал "полную стоимость" сложных алгоритмов. Они более склонны к ошибкам в первоначальной реализации, а кроме этого, последующим кураторам труднее их понять.

Другим важным практическим принципом является создание руководств хакеров. Для дистрибутивов исходного кода всегда было полезным включение документации, неформально описывающей ключевые структуры данных и алгоритмы кода. Действительно, Unix-программисты часто лучше создают хакерские руководства, чем пишут документацию для конечных пользователей.

Сообщество открытого исходного кода постигло и выработало этот обычай. Кроме того что хакерские руководства являются рекомендациями для будущих кураторов, в проектах с открытым исходным кодом они также предназначены для того, чтобы способствовать добавлению функций и исправлению ошибок со стороны программистов-любителей. Характерным примером является файл Design Notes, поставляемый с программой

fetchmail . Исходные коды ядра Linux включают в себя буквально десятки подобных документов.

В главе 19 описаны соглашения, выработанные Unix-разработчиками для облегчения изучения дистрибутивов исходных кодов и создания исполняемого кода.

7

Мультипрограммирование: разделение процессов для разделения функций

Если мы придаем большое значение структурам данных, то мы должны придавать большое значение независимой (и, следовательно, одновременной) обработке. Иначе для чего мы собираем объекты в структуру? Почему мы терпим языки, которые дают нам одно без другого?

Раздел Epigrams in Programming в журнале ACM SIGPLAN (17 #9, 1982) —Алан Перлис (Alan Perlis)

Наиболее характерной методикой разбиения программ на модули в операционной системе Unix является разделение крупных программ на множество взаимодействующих процессов. Обычно в мире Unix данная методика называется "многопроцессорной обработкой" (multiprocessing), но в этой книге, для того чтобы избежать путаницы с многопроцессорным аппаратным обеспечением, используется более ранний термин "мультипрограммирование" (multiprogramming).

Мультипрограммирование является "особенно туманной" областью проектирования, в которой реализовано несколько основополагающих принципов хорошей практики. Многие программисты, превосходно разбирающиеся в том, как разбивать код на подпрограммы, тем не менее, в итоге пишут целые приложения как массивные однопроцессные монолиты, которые разрушаются под тяжестью своей собственной внутренней сложности.

В Unix-проектировании подход "решать одну задачу хорошо" применяется на уровне взаимодействующих программ подобно тому, как во взаимодействующих подпрограммах он применяется внутри программы. Особое значение придается мелким программам, соединенным четко определенным методом межпроцессного обмена данными или совместно используемыми файлами. Соответственно, операционная система Unix побуждает программистов разбивать создаваемые программы на более простые подпроцессы и уделять внимание интерфейсам между этими подпроцессами. Такой подход система обеспечивает тремя основными способами:

• малозатратное создание подпроцессов;

• предоставление методов, которые относительно упрощают обмен данными между процессами (вызовы с созданием подоболочки, перенаправление ввода/вывода, каналы, передача сообщений и сокеты);

• поддержка простых, прозрачных, текстовых форматов данных, которые могут передаваться посредством каналов и сокетов.

Малозатратное создание дочерних процессов и простое управление процессами являются весьма важными факторами для Unix-стиля программирования. В такой операционной системе, как VAX VMS, где запуск процессов является дорогой и медленной операцией, требующей специальных привилегий, программисты вынуждены создавать массивные монолиты, поскольку не имеют другого выбора. К счастью, семейство Unix отличается направленностью в сторону более низких издержек

fork(2), а не в сторону более высоких. В частности, операционная система Linux особенно эффективна в этом отношении, подпроцессы создаются в ней быстрее, чем возникают параллельные процессы во многих других операционных системах[63].

Исторические причины подталкивают многих Unix-программистов мыслить понятиями множества взаимодействующих процессов по опыту shell-программирования. Оболочка

относительно упрощает создание групп из множества соединенных каналами процессов, запущенных в приоритетном или фоновом режиме или с одновременным использованием обоих режимов.

В оставшейся части данной главы рассматриваются последствия малозатратного создания подпроцессов, а также описывается, как и когда применять каналы, сокеты и другие методы межпроцессного взаимодействия (IPC), для того чтобы разделить конструкцию на взаимодействующие процессы. (В следующей главе философия разделения функций применяется к проектированию интерфейсов.)

Тогда как выгода от разбиения программ на взаимодействующие процессы заключается в снижении глобальной сложности, затраты такого подхода связаны с необходимостью уделять больше внимания разработке протоколов, которые используются для передачи информации и команд между процессами. (В программных системах всех видов ошибки скапливаются в интерфейсах.)

В главе 5 рассматривается самый низкий уровень данной проблемы проектирования — каким образом располагать протоколы прикладного уровня, которые являются прозрачными, гибкими и расширяемыми. Однако эта проблема имеет второй, более высокий уровень, который в главе 5 не рассматривался, — это проектирование конечных автоматов для каждой стороны информационного обмена.

Не сложно применить хороший стиль к синтаксису протокола прикладного уровня таких моделей, как SMTP, BEEP или XML-RPC. Реальная сложность заключается не в синтаксисе протокола, а в его

логике — проектировании протокола, который одновременно является достаточно выразительным и не имеет проблем

взаимоблокировки процессов. Почти так же важно то, что протокол должен быть

видимым , для того чтобы быть выразительным и свободным от взаимоблокировки. Программисты, пытающиеся мысленно моделировать поведение взаимодействующих программ и проверять его корректность, должны иметь возможность поступать именно так.

Следовательно, в обсуждении данных проблем основное внимание уделено видам логики протоколов, каждый из которых программист свободно использует для каждого вида межпроцессного взаимодействия.

7.1. Отделение контроля сложности от настройки производительности

Прежде всего, необходимо избавиться от некоторых ложных целей. В данной главе обсуждение

не касается использования одновременных операций для повышения производительности. Рассмотрение этой идеи до того как будет сформулирована ясная структура, которая сводит к минимуму глобальную сложность, является преждевременной оптимизацией, т.е. корнем всех зол (дальнейшее обсуждение приведено в главе 12).

Близкой ложной целью являются параллельные процессы (threads) (т.е. множество одновременно выполняемых процессов, совместно использующих одно адресное пространство памяти). Использование параллельных процессов — средство повышения производительности. Для того чтобы не уходить далеко от основной линии обсуждения,

данная методика более подробно рассматривается в конце данной главы. Здесь достаточно сделать обобщение: параллельные процессы не уменьшают глобальную сложность, а скорее увеличивают ее, и поэтому следует избегать их использования, кроме случаев крайней необходимости.

С другой стороны, соблюдение правила модульности

не является ложной целью, поскольку модульность позволяет упростить программы и, следовательно, работу программиста. Все причины для разделения процессов являются продолжением причин для разделения модулей, которое рассматривалось в главе 4.

Другой важной причиной разбиения программ на взаимодействующие процессы является повышение безопасности. В операционной системе Unix программы, которые запускаются обычными пользователями, но должны иметь доступ к важным с точки зрения безопасности системным ресурсам, получают такой доступ с помощью функции

setuid [64]. Исполняемые файлы представляют собой наименьший элемент кода, который может содержать setuid-бит. Следовательно, каждая строка кода в исполняемом setuid-файле должна быть "благонадежной". (Вместе с тем хорошо написанные setuid-программы, сначала предпринимают все необходимые привилегированные действия, а на последующих этапах своей работы понижают свои привилегии до уровня пользователя.)

Обычно привилегии setuid-программы требуются для выполнения ею одной или нескольких операций. Часто имеется возможность разбить такую программу на взаимодействующие процессы: мелкий, не требующий использования функции setuid, и более крупный, который в ней не нуждается. Если такое разделение возможно, то "благонадежным" должен быть только код в меньшей программе. В значительной степени благодаря подобному разделению и делегированию функций, операционная система Unix обладает большими достижениями в обеспечении безопасности[65], чем ее конкуренты.

7.2. Классификация IPC-методов в Unix

Как и в однопроцессных структурах программ, простейшая организация является наилучшей. В оставшейся части данной главы представлены IPC-методики приблизительно в порядке возрастания сложности их программирования. Прежде чем использовать более сложные методики; следует с помощью прототипов и эталонных тестов получить доказательства того, что простые методики не подходят. Такой подход часто приводит к поразительным результатам.

7.2.1. Передача задач специализированным программам

В простейшей форме взаимодействия программ, которая возможна благодаря малозатратному созданию дочерних процессов, одна программа вызывает другую для решения специализированной задачи. Поскольку вызванная программа часто задается как команда оболочки Unix через вызов

system(3), данная операция часто называется

вызовом программы с созданием подоболочки (shelling out) . Вызываемая программа

наследует управление клавиатурой и пользовательским дисплеем и выполняется до завершения. Когда она прекращает свою работу, вызывающая программа возобновляет управление клавиатурой и дисплеем и продолжает выполнение[66]. Поскольку вызывающая программа не обменивается данными с вызванной программой во время работы последней, конструкция протокола не является проблемой при таком виде взаимодействия, кроме очевидного случая, при котором вызывающая программа может для изменения поведения вызванной программы передать ей аргументы командной строки.

Классическим случаем вызова с созданием подоболочки в Unix является вызов редактора из программы чтения почты или новостей. Разработчик, придерживающийся традиций Unix,

не встраивает специально созданный редактор в программу, которая требует обычного текстового ввода. Вместо этого программист предоставляет пользователю возможность указать предпочтительный редактор, который будет вызываться при необходимости.

Специализированная программа обычно сообщается с родительским процессом через файловую систему, считывая или модифицируя файл (или файлы) с определенным расположением. Так работают вызываемые редакторами и почтовыми агентами программы.

В распространенном варианте данной модели специализированная программа может получать данные на свой стандартный ввод и вызываться с помощью функции C-библиотеки popen(..., "w") или как часть сценария оболочки. Или она может отправлять данные на свой стандартный вывод и вызываться с помощью функции popen(..., "r") или как часть сценария оболочки. (Если программа считывает стандартный ввод и записывает данные в стандартный вывод, то она выполняет эти операции в пакетном режиме, завершая все операции чтения до записи каких-либо данных.) Данный вид дочерних процессов обычно не называют процессами подоболочки. Для их обозначения не существует стандартного термина, однако такие программы можно называть "прикрепляемыми".

Ключевым моментом во всех описанных случаях является то, что специализированные программы во время работы не обмениваются данными с родительскими. Они имеют связанный протокол только в том случае, когда какая-либо программа (главная или подчиненная), принимающая ввод от другой, должна быть способна осуществлять синтаксический анализ ввода.

7.2.1.1. Учебный пример: пользовательский почтовый агент

mutt

Пользовательский почтовый агент

mutt является современным представителем наиболее важной традиции проектирования программ для обработки электронной почты в Unix. Данная программа имеет простой экранный интерфейс с одноклавишными командами для просмотра и чтения почты.

Если

mutt используется для создания почтовых сообщений (либо если данная программа вызвана с адресом в качестве аргумента командной строки или с помощью одной из команд для создания ответного сообщения), то программа определяет значение переменной EDITOR, а затем генерирует имя временного файла. Значение данной переменной используется как команда, а имя временного файла как ее аргумент[67]. Когда запущенная таким образом

программа прекращает свою работу,

mutt возобновляет управление, предполагая, что временный файл содержит необходимый текст сообщения.

Данное соглашение в Unix соблюдается почти во всех программах создания почтовых сообщений и сообщений в группах новостей. И вследствие этого программистам, реализующим такие программы, не требуется писать сотни неизбежно различающихся редакторов, а пользователям не требуется изучать сотни различных интерфейсов. Вместо этого пользователи могут использовать с такими программами предпочтительные редакторы.

Важным вариантом такой стратегии является вызов с созданием подоболочки небольшой программы-посредника, передающей специальное задание уже запущенному экземпляру большой программы, такой как редактор или Web-браузер. Поэтому разработчики, на X-дисплеях которых обычно уже имеется запущенный экземпляр редактора

emacs , могут установить переменную EDITOR=emacsclient и в случае необходимости редактировать сообщение в

mutt открывать данные буфера в

emacs . Целью данного подхода является не экономия памяти или других ресурсов, а предоставление пользователю возможности объединять все редактирование в одном

emacs- процессе (поэтому, например, при вырезании и вставке между буферами можно было переносить внутренние параметры

emacs, такие как выделение шрифта).

7.2.2. Каналы, перенаправление и фильтры

После Кена Томпсона и Денниса Ритчи одной из наиболее важных фигур в истории создания Unix был, вероятно, Дуг Макилрой. Созданная им конструкция

канала (pipe) в той или иной форме влияла на конструкцию операционной системы Unix, поддерживая зарождающуюся в ней философию "хорошего решения одной задачи" и способствуя большинству более поздних форм IPC в Unix (в частности, абстракции сокетов, применяемой для поддержки сетей).

Работа каналов определяется соглашением, согласно которому каждая программа изначально имеет доступные ей (по крайней мере) два потока данных ввода-вывода: стандартный ввод и стандартный вывод (числовые дескрипторы файлов 0 и 1 соответственно). Многие программы могут быть написаны в виде

фильтров (filters) , которые последовательно считывают данные со стандартного ввода и записывают только в стандартный вывод.

Обычно такие потоки подключены к пользовательской клавиатуре и дисплею соответственно. Однако оболочки в операционной системе Unix обеспечивают универсальную поддержку операций

перенаправления (redirection) , которые подключают стандартный ввод и стандартный вывод

к файлам. Поэтому команда

ls >foo

отправляет вывод команды

ls(1) в файл с именем "foo". С другой стороны, команда

wc <foo

вынуждает утилиту для подсчета слов

wc(1) принять на свой стандартный ввод данные из файла "foo" и отправить сведения о количестве символов/слов/строк на стандартный вывод.

Канал подключает стандартный вывод одной программы к стандартному вводу другой. Цепочка программ, соединенных таким способом, называется

конвейером (pipeline) . Команда

ls | wc

позволяет получить количество символов/слов/строк в списке файлов текущего каталога. (В данном случае, вероятно, действительно полезным будет только количество строк.)

Одним излюбленным конвейером был "bc | speak" — "говорящий" калькулятор.

Он "знал" названия чисел до вигинтеллиона (1063) Дуг Макилрой.

Важно отметить, что все этапы конвейера работают одновременно. Каждый этап ожидает ввода на выходе из предыдущего этапа, но ни один этап не должен завершить работу до того, как следующий получит возможность запуститься. Важность этого свойства отмечена далее при рассмотрении интерактивного использования таких конвейеров как, например, отправка длинного вывода какой-либо команды утилите

more(1) .

Легко недооценить силу комбинирования каналов и перенаправления. В качестве полезного примера в работе

"The Unix Shell As a 4GL" [75] показано, как, используя данные средства в качестве каркаса, можно скомбинировать несколько простых утилит, чтобы обеспечить поддержку создания и изменения реляционных баз данных, выраженных в виде простых текстовых таблиц.

Основной недостаток каналов заключается в том, что они являются однонаправленными. Для компонента конвейера не существует другой возможности отправить управляющую информацию обратно в канал, кроме прерывания (в этом случае предыдущий этап получает сигнал SIGPIPE на следующей операции записи). Соответственно, протоколом для передачи данных является просто формат ввода принимающего этапа.

Выше были описаны неименованные каналы, создаваемые оболочкой. Существует их разновидность,

именованный канал (named pipe) , который представляет собой особый вид файла. Если две программы открывают файл, одна для чтения и другая для записи, то именованный канал действует как соединительный элемент между ними. Именованные каналы — почти пережиток истории. Они почти вытеснены именованными

сокетами, которые описываются ниже. (История этого реликта подробнее описана в разделе "System V IPC".)

7.2.2.1. Учебный пример: создание канала к пейджеру

Существует множество вариантов использования конвейеров. Например, Unix-утилита

ps(1) выводит на стандартный вывод список процессов, "не заботясь" о том, что верхняя часть длинного листинга может не поместиться на пользовательском дисплее и исчезнет слишком быстро, чтобы пользователь успел ее увидеть. В операционной системе Unix имеется другая программа,

more(1) , которая отображает данные, полученные на стандартный ввод, блоками, размеры которых не превышают размеры экрана, и после отображения каждого блока ожидает нажатия клавиши пользователем.

Таким образом, если пользователь вводит команду "ps | more", передавая вывод утилиты

ps(1) на ввод

more(1), на экране последовательно после каждого нажатия клавиши будут отображаться страницы списка процессов.

Подобная возможность комбинировать программы является чрезвычайно полезной. Но действительный выигрыш в данном случае не сводится к изящным комбинациям. Именно благодаря тому, что существуют каналы и программа

more(1), другие программы могут быть проще. Использование каналов означает, что в таких программах, как

ls(1) (и других, записывающих данные в стандартный вывод), не требуется культивировать собственные средства постраничного вывода (пейджеры), а пользователи избавлены от тысяч встроенных пейджеров (каждый из которых, естественно, обладает собственными особенностями применения). Благодаря каналам, предотвращается раздувание кода и сокращается глобальная сложность.

В дополнение к этому, если потребуется настроить режим работы пейджера, то это можно сделать в

одном месте путем изменения

одной программы. Действительно, может существовать множество пейджеров и все они будут полезны для каждого приложения, которое записывает информацию в стандартный вывод.

Фактически дело обстоит именно так. В современных Unix-системах

more(1) почти полностью заменена утилитой

less(1) , в которой добавлена возможность просматривать страницы отображенного файла не только сверху вниз, но и снизу вверх[68]. Ввиду того, что

less(1) отделена от использующих ее программ, существует возможность просто связать ее псевдонимом с "more" в оболочке, установить значение переменной среды PAGER равным

"less" (см. главу 10) и получить все преимущества лучшего пейджера со всеми Unix-программами, написанными соответствующим образом.

7.2.2.2. Учебный пример: создание списков слов

Более интересным является пример, в котором программы, объединенные в конвейер, взаимодействуют в целях трансформации данных, для реализации которой в других менее гибких средах потребовалось бы писать специальный код.

Рассмотрим следующий конвейер

tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'

Первая команда преобразовывает не алфавитно-цифровые символы, полученные на стандартном вводе, в разделители строк на стандартном выводе. Вторая команда сортирует строки из стандартного ввода и записывает отсортированные данные в стандартный вывод, исключая все, кроме одной копии из диапазона идентичных смежных строк. Третья команда удаляет все строки, состоящие исключительно из цифр. Вместе данные команды генерируют на стандартном выводе отсортированный список слов из текста, полученного на стандартном вводе.

7.2.2.3. Учебный пример:

pic2graph

Исходный shell-код для программы

pic2graph(1) поставляется вместе с пакетом инструментальных средств для форматирования текстов

groff , созданным Фондом свободного программного обеспечения. Данная программа преобразовывает диаграммы, написанные на языке PIC, в растровые изображения. В примере 7.1 показан конвейер, находящийся в главной части кода.Пример 7.1. Конвейер

pic2graph

(echo ".EQ"; echo $eqndelim; echo ".EN"; echo ".PS"; cat; echo ".PE")|\

 groff -e -p $groffpic_opts -Tps >${tmp}.ps \

 && convert -crop 0x0 $convert_opts ${tmp}.ps ${tmp}.${format} \

 && cat ${tmp}.${format}

Реализация программы

pic2graph иллюстрирует то, как много способен сделать один конвейер, просто вызывая уже имеющиеся инструменты. Работа программы начинается с преобразования входных данных в соответствующую форму. Затем полученные данные обрабатываются

groff(1) для создания PostScript-представления. На завершающей стадии PostScript конвертируется в растровое изображение. Все описанные детали скрыты от пользователя, который просто видит то, как в программу с одной стороны поступает исходный PIC-код, а с другой стороны из нее выходит растровое изображение, готовое для включения в Web-страницу.

Данный пример примечателен тем, что он иллюстрирует способность каналов и фильтров адаптировать программы для неожиданного применения. Программа, интерпретирующая PIC-код,

pic(1) , первоначально разрабатывалась только для внедрения диаграмм в форматированные документы. Большинство остальных программ в данной инструментальной связке были частью почти отжившей в настоящее время конструкции. Однако PIC остается удобным языком для нового применения, такого как описание диаграмм, встраиваемых в HTML-документы. Он получил право на существование, поскольку инструменты, подобные

pic2graph(1), способны связывать все механизмы, необходимые для преобразования вывода утилиты

pic(1) в более современный формат.

Программа

pic(1) подробнее рассматривается в главе 8 при обсуждении конструкций мини-языков.

7.2.2.4. Учебный пример: утилиты

bc(1) и

dc(1)

Частью классического инструментального набора, происходящего из Unix Version 7, является пара программ-калькуляторов. Программа

dc(1) представляет собой простой калькулятор, принимающий на стандартный ввод текстовые строки, состоящие из обратных польских записей (Reverse-Polish Notation — RPN), и отправляющий результаты вычислений на стандартный вывод. Программа

bc(1) допускает более сложный, инфиксный синтаксис, который подобен традиционной алгебраической форме записи. Кроме того, данная программа способна задавать и считывать значения переменных и определять функции для сложных формул.

Хотя современная GNU-реализация

bc(1) является автономной, ее классическая версия передавала команды в программу

dc(1) посредством канала. В этом разделении труда утилита

bc(1) осуществляет подстановку значений переменных, разложение функций и преобразование инфиксной записи в обратную польскую, но сама, по существу, не выполняет вычислений. Вместо этого результаты RPN-преобразования входных выражений для расчета передаются программе

dc(1) .

Такое разделение функций имеет очевидные преимущества. Это означает, что пользователям приходится выбирать предпочтительную форму записи, но дублировать логику для числовых расчетов с произвольной точностью (умеренно сложную) не требуется. Каждая из двух программ может быть менее сложной, чем один калькулятор с выбором формы записи. Отладку и мысленное моделирование можно осуществлять независимо для каждого компонента.

В главе 8 данные программы рассматриваются в несколько другом свете, как узкоспециальные мини-языки.

7.2.2.5. Контрпример: почему программа

fetchmail не выполнена в виде конвейера

В понятиях Unix

fetchmail является неудобной большой программой, изобилующей различными параметрами. Рассматривая способ транспортировки почты, можно предположить, что данную программу можно было бы разложить на компоненты конвейера. Предположим, что она разбита на ряд программ: несколько программ доставки для получения почты с POP3- и IMAP-узлов, и локальный SMTP-инжектор. Конвейер мог бы передавать почтовый формат Unix. Существующую сложную конфигурацию

fetchmail можно было бы заменить сценарием оболочки, содержащим строки команд. В такой конвейер можно также добавить фильтры для блокировки спама.

#!/bin/sh

imap jrandom@imap.ccil.org | spamblocker | smtp jrandom

imap jrandom@imap.netaxs.com | smtp jrandom

pop ed@pop.tems.com | smtp jrandom

Такая конструкция была бы весьма изящной и соответствовала бы духу Unix, но не смогла бы работать. Причина рассматривалась выше — конвейеры являются однонаправленными.

Одной из функций программы доставки (

imap или

pop) было бы принятие решения о том, отправлять ли запрос на удаление каждого принимаемого сообщения. В существующей организации

fetchmail отправка такого запроса POP- или IMAP-серверу может быть задержана до тех пор, пока программа не получит подтверждения о том, что локальный SMTP-слушатель взял на себя ответственность за данное сообщение. Версия программы, организованная в виде конвейера из небольших компонентов, потеряла бы данное свойство.

Рассмотрим, например, последствия аварийного завершения smtp-инжектора вследствие того, что SMTP-получатель сообщил о переполнении диска. Если программа доставки уже удалила почту, сообщения будут утеряны. Это означает, что программа доставки не может удалять почту до тех пор, пока не получит соответствующее уведомление от

smtp- инжектора. Причем с данной проблемой связан ряд вопросов. Каким образом программы обменивались бы данными? Какое в точности сообщение было бы возвращено инжектором? Общая сложность такой системы и ее подверженность неочевидным ошибкам были бы выше, чем сложность монолитной программы.

Конвейеры являются превосходными инструментами, но они не универсальны.

7.2.3. Упаковщики

Противоположностью вызова с созданием подоболочки является

упаковщик (wrapper). Упаковщик создает новый интерфейс для вызываемой программы или определяет его. Часто упаковщики используются для сокрытия деталей сложных shell-конвейеров. Упаковщики интерфейсов обсуждаются в главе 11. Наиболее специализированные упаковщики являются достаточно простыми и, тем не менее, весьма полезными.

Как и в случае вызова с созданием подоболочки, связанного протокола не существует, поскольку программы не обмениваются данными во время выполнения вызываемой программы. Однако обычно упаковщик существует для указания аргументов, изменяющих поведение этой программы.

7.2.3.1. Учебный пример: сценарии резервного копирования

Специализированные упаковщики представляют собой классический пример использования Unix shell и других языков сценариев. Одним из распространенных и характерных видов специализированных упаковщиков является сценарий резервного копирования. Он может быть однострочным, таким же простым, как приведенный ниже.

```
tar -czvf /dev/st0 "$@"
```

Приведенная выше команда является упаковщиком для утилиты архивирования

tar(1) , который предоставляет один фиксированный аргумент (накопитель на магнитных лентах /dev/st0) и передает

tar все остальные аргументы, указанные пользователем ("$@")[69].

7.2.4. Оболочки безопасности и цепи Бернштайна

Один из распространенных способов использования сценариев упаковщиков — это создание

оболочек безопасности (security wrappers) . Сценарий безопасности может вызывать программу-диспетчер (gatekeeper) для проверки мандата (credential), а затем в зависимости от значения, возвращенного программой-диспетчером, запустить другую программу.

Образование цепей Бернштайна (Bernstein chaining) представляет собой специализированную методику применения оболочек безопасности, впервые разработанную Даниелем Бернштайном (Daniel J. Bernstein), который задействовал ее в многих своих пакетах. (Подобная модель наблюдается в таких командах, как

nohup(1) и

su(1) , но условность выполнения отсутствует.) Концептуально цепи Бернштайна подобны конвейерам, но, в отличие от последних, в цепях каждый успешный этап заменяет предыдущий, а не выполняется одновременно с ним.

Обычным применением данной методики является заключение привилегированных приложений в некоторой программе-диспетчере, которая затем может передать параметры менее привилегированной программе. В данной методике несколько программ объединяются с помощью вызовов exec или, возможно, комбинации вызовов fork и exec. Все программы указываются в одной командной строке. Каждая программа осуществляет некоторую функцию и (в случае успешного завершения) запускает

exec(2) для остальной части командной строки.

Пакет Бернштайна

rblsmtpd является принципиальным примером. Он служит для поиска узла в антиспамной DNS-зоне системы предотвращения некорректного использования почты (Mail Abuse Prevention System). Свои функции данный пакет выполняет путем отправки DNS-запроса на IP-адрес, переданный ему как значение переменной среды TCPREMOTEIP. Если запрос оказался успешным, то

rblsmtpd запускает собственный SMTP-сервер, который отклоняет почту. В противном случае предполагается, что оставшиеся аргументы командной строки вводят в действие агент доставки почты, который поддерживает протокол SMTP, и передаются для запуска

exec(2) .

Еще один пример можно встретить в пакете

qmail Бернштайна. Пакет содержит программу, которая называется

condredirect . Первым параметром является адрес электронной почты, остальные параметры — программа-диспетчер и аргументы,

condredirect разветвляется и запускает программу-диспетчер со своими аргументами. В случае успешного завершения программой-диспетчером своей работы,

condredirect перенаправляет почтовое сообщение, ожидающее в stdin, на указанный адрес электронной почты. В этом случае, в противоположность

rblsmtpd , решение принимается дочерним процессом. Этот случай несколько больше походит на классический вызов с созданием подоболочки.

Более сложным примером является POP3-сервер

qmail . Он состоит из трех программ:

qmail-popup, checkpassword и

qmail-pop3d . Программа

Checkpassword взята из отдельного пакета, остроумно названного

checkpassword, и, естественно, предназначена для проверки паролей. В протоколе POP3 предусмотрен этап аутентификации и этап работы с почтовым ящиком. Перейдя к этапу работы с почтовым ящиком, пользователь не имеет возможности вернуться к этапу аутентификации. Это идеальное применение для цепей Бернштайна.

Первым параметром программы

qmail-popup является имя узла для использования в POP3-приглашениях. Остальные ее параметры, после того как имя пользователя POP3 и пароль были доставлены, разделяются и передаются

exec(2) . Если данная программа возвращает отказ, то пароль, вероятнее всего, не верный,

qmail-popup сообщает об этом и ожидает ввода другого пароля. В противном случае предполагается, что программа окончила POP3-диалог, поэтому

qmail-popup завершает работу.

Ожидается, что программа, указанная в командной строке

qmail-popup , считывает 3 строки, ограниченные символом NULL, из дескриптора файла 3[70]: имя пользователя, пароль и ответ на криптографический запрос, если он есть. В настоящее время такой программой является

checkpassword , которая принимает в качестве параметров имя

qmail-pop3d и его параметры. Программа

checkpassword завершается с ошибкой в том случае, если был введен несоответствующий пароль. В противном случае она принимает пользовательские идентификаторы uid и gid, домашний каталог пользователя и выполняет оставшуюся часть командной строки от имени данного пользователя.

Создание цепей Бернштайна полезно в ситуациях, когда приложение требует setuid- или setgid-привилегий для инициализации соединения или для получения какого-либо мандата, а затем понижает эти привилегии, чтобы можно было использовать последующий вовсе не обязательно "благонадежный" код. Поскольку дочерняя программа была запущена с помощью вызова exec, она не может установить свой действительный идентификатор пользователя равным идентификатору администратора. Такая методика является более гибкой, чем использование одного процесса, поскольку можно модифицировать поведение системы путем внедрения в цепочку другой программы.

Например, программа

rblsmtpd (упомянутая выше) может быть вставлена в цепь Бернштайна между программой tcpserver (из пакета

ucspi-tcp) и реальным SMTP-сервером, обычно

qmail-smtpd. В то же время она также работает с

inetd(8) и sendmail -bs.

7.2.5. Подчиненные процессы

Иногда дочерние программы интерактивно принимают и возвращают данные вызвавшим их программам через каналы, связанные со стандартным выводом и вводом. В отличие от простых вызовов с созданием подоболочки и конструкций, которые выше были названы "прикрепляемыми", как главный, так и подчиненный процессы нуждаются в наличии внутренних конечных автоматов для обработки протокола между ними без взаимоблокировки или конкуренции. Такая организация является значительно более сложной и более трудной в отладке, чем простые вызовы с созданием подоболочки.

Вызов

popen(3) в Unix способен устанавливать либо канал ввода, либо канал вывода для подоболочки, но не оба канала для подчиненного процесса — видимо, для поощрения более простого программирования. И фактически интерактивный обмен данными между главным и подчиненным процессом является настолько сложным, что обычно используется только в ситуациях, когда (a) связанный протокол является крайне примитивным, либо (b) подчиненный процесс спроектирован для обмена данными по протоколу прикладного уровня в соответствии с принципами, которые описывались в главе 5. Данная проблема и способы ее разрешения рассматриваются в главе 8.

При написании пары "главный-подчиненный" хорошей практикой считается заставить главный процесс поддерживать ключ командной строки или переменную среды, которая позволяет вызывающим программам устанавливать собственную подчиненную команду. Кроме прочего, это полезно для отладки. Нередко обнаруживается удобство такой конструкции во время разработки для вызова реального подчиненного процесса из тестовой программы, которая осуществляет мониторинг и протоколирование транзакций между главным и подчиненным процессом.

Если выясняется, что взаимодействие главных и подчиненных процессов в разрабатываемой программе становится нетривиальным, то, возможно, следует задуматься о переходе к более равноправной организации, используя такие методики, как сокеты или общая память.

7.2.5.1. Учебный пример:

scp и

ssh

Индикаторы выполнения — один распространенный случай, в котором связанный протокол действительно является тривиальным. Утилита

scp(1) (secure-copy command — команда безопасного копирования) вызывает программу

ssh(1) как подчиненный процесс, перехватывая со стандартного вывода ssh достаточно информации для того, чтобы переформатировать отчеты в виде ASCII-анимации индикатора выполнения[71].

7.2.6. Равноправный межпроцессный обмен данными

Все рассмотренные выше методы обмена данными имеют некоторую неявную иерархию, в которой одна программа фактически контролирует или управляет другой, а в противоположном направлении сведения обратной связи не передаются или передаются в ограниченном количестве. В системах связи или сетях часто требуется создание

равноправных (peer-to-peer) каналов, обычно (но не обязательно) поддерживающих свободную передачу данных в обоих направлениях. Ниже рассматриваются методы равноправного обмена данными, а несколько учебных примеров рассматривается в последующих главах.

7.2.6.1. Временные файлы

Использование временных файлов в качестве буферов обмена данными является старейшей из существующих IPC-методик. Несмотря на недостатки, она остается удобной в сценариях командных интерпретаторов и одноразовых программах, где более сложный и координированный метод обмена данными был бы излишним.

Наиболее очевидная проблема при использовании временных файлов в качестве IPC-методики заключается в мусоре, который остается в файловой системе, если обработка была прервана до того, как временный файл можно было удалить. Менее очевидный риск связан с коллизиями между несколькими экземплярами программы, использующими одно и то же имя временного файла. Именно поэтому для shell-сценариев является традиционным включение shell-переменной $$ в имена создаваемых ими временных файлов. В данной переменной содержится идентификатор процесса оболочки, и ее использование действительно гарантирует, что имя файла будет уникальным (такой же технический прием поддерживается в языке Perl).

Наконец, если атакующий знает расположение записываемого временного файла, то может переписать его и, вероятно, считать данные создавшего этот файл процесса или "обмануть" использующий его процесс путем внедрения в файл модифицированных или фиктивных данных[72]. Это рискованно с точки зрения безопасности, а если задействованные процессы обладают привилегиями администратора, то риск представляется весьма серьезным. Его можно уменьшить с помощью тщательной настройки полномочий на каталог временных файлов, однако известно, что данные мероприятия, вероятно, приводят к утечкам.

Все описанные проблемы остаются в стороне, временные файлы до сих пор занимают собственную нишу, поскольку они легко устанавливаются, они являются гибкими и менее подверженными взаимоблокировкам и конкуренции, чем более сложные методы. Иногда другие методы просто не подходят. Соглашения о вызовах дочернего процесса могут потребовать передачи файла для выполнения над ним операций. Первый пример вызова редактора с созданием подоболочки демонстрирует это в полной мере.

7.2.6.2. Сигналы

Самый простой и грубый способ сообщения между двумя процессами на одной машине заключается в том, что один из них отправляет другому какой-либо

сигнал (signal) . Сигналы в операционной системе Unix представляют собой форму

программного прерывания. Каждый сигнал характеризуется стандартным влиянием на получающий его процесс (обычно процесс уничтожается). Процессом может быть объявлен

обработчик сигналов (signal handler) , который подменяет их стандартные действия. Обработчик представляет собой функцию, которая выполняется асинхронно при получении сигнала.

Первоначально сигналы были встроены в Unix не как средство IPC, а как способ, позволяющий операционной системе сообщать программам об определенных ошибках и критических событиях. Например, сигнал SIGHUP отправляется каждой программе, запущенной из определенного терминального сеанса, когда этот сеанс завершается. Сигнал SIGINT отправляется любому процессу, подключенному в текущий момент времени к клавиатуре, когда пользователь вводит определенный символ прерывания (часто control-C). Тем не менее, сигналы могут оказаться полезными в некоторых IPC-ситуациях (и в набор сигналов стандарта POSIX включено два сигнала, предназначенных для таких целей, SIGUSR1 и SIGUSR2). Часто они используются как канал управления для

демонов (daemons) (программы, которые работают постоянно и невидимо, т.е. в фоновом режиме). Это способ для оператора или другой программы сообщить демону о том, что он должен повторно инициализироваться, выйти из спящего режима для выполнения работы или записать в определенное место сведения о внутреннем состоянии или отладочную информацию.

Я настаивал на том, чтобы сигналы SIGUSR1 и SIGUSR2 были созданы для BSD. Люди хватались за системные сигналы, чтобы заставить их делать то, что им нужно для IPC, так что (например) некоторые программы, которые аварийно завершались в результате ошибки сегментации, не создавали дамп памяти, потому что сигнал SIGSEGV был модифицирован.

Это общий принцип — люди будут хотеть модифицировать любые создаваемые вами инструменты. Поэтому необходимо проектировать программы так, чтобы их либо нельзя было модифицировать, либо можно было модифицировать аккуратно. Это единственные варианты. За исключением, конечно, того случая, когда программу проигнорируют — весьма надежный способ остаться "незапятнанным", однако он менее удовлетворительный, чем может показаться на первый взгляд. Кен Арнольд.

Методика, которая часто применяется с сигнальным IPC, также называется

pid-файлом . Программы, которым требуется получать сигналы, записывают небольшие файлы, содержащие идентификатор процесса или PID (process ID), в определенный каталог (часто /var/run или домашний каталог запускающего программу пользователя). Другие программы могут считывать данный файл для определения PID. PID-файл также может служить в качестве неявного

файла блокировки (lock file) в случаях, когда необходимо запустить одновременно не более одного экземпляра демона.

Фактически существует две различные разновидности сигналов. В ранних реализациях (особенно в V7, System III и в ранней System V) обработчик для определенного сигнала каждый раз после срабатывания переустанавливается в стандартное состояние. Следовательно, в результате двух одинаковых сигналов, отправленных быстро друг за другом, процесс обычно уничтожается независимо от того, какой обработчик был установлен.

Версии 4.x BSD Unix перешли к использованию "надежных" сигналов, которые не переустанавливаются, если пользователь не требует этого явно. Также в данных версиях были представлены примитивы для блокировки или временной приостановки обработки определенного набора сигналов. В современных Unix-системах поддерживается оба стиля.

Для нового кода следует использовать непереустанавливаемые точки входа в BSD-стиле, однако в случае если код когда-либо будет переноситься в реализацию, которая не поддерживает их, необходимо использовать методику "безопасного программирования".

Получение N сигналов не обязательно N раз вызывает обработчик сигналов. В старой модели сигналов System V два или более сигнала, поданные очень близко (т.е. в одном кванте времени целевого процесса), могут привести к различным проявлениям конкуренции[73] или аномалиям. В зависимости от варианта семантики сигналов, который поддерживается в системе, второй и последующие экземпляры могут игнорироваться, вызывать неожиданное завершение процесса или задерживаться, пока обрабатываются предыдущие экземпляры (в современных Unix-системах последней вариант наиболее вероятен).

Современный API-интерфейс сигналов переносится на все последние версии Unix, но не на Windows или классическую (предшествующую OS X) MacOS.

7.2.6.3. Системные демоны и традиционные сигналы

Многие широко известные системные демоны в качестве сигнала для повторной инициализации (т.е. перезагрузки их конфигурационных файлов) принимают сигнал SIGHUP (первоначально данный сигнал отправлялся программам при разрыве последовательной линии, например при разрыве модемного соединения). Примеры включают в себя Apache и Linux-реализации таких демонов, как

bootpd(8), gated(8), inetd(8), mountd(8), named(8), nfsd(8) и

ypbind(8). В некоторых случаях сигнал SIGHUP принимается "в его первоначальном смысле", как сигнал разрыва сеанса (особенно в Linux-реализации

pppd(8)), но эта роль в настоящее время, как правило, отводится сигналу SIGTERM.

SIGTERM (terminate — завершить) часто принимается как сигнал постепенного завершения (чем он отличается от SIGKILL, который выполняет немедленное уничтожение и не может быть блокирован или перехвачен). Данный сигнал часто вызывает очистку временных файлов, удаление последних изменений в базах данных и подобные действия.

При написании демонов рекомендуется придерживаться правила наименьшей неожиданности, т.е. использовать данные соглашения и читать справочные руководства для поиска существующих моделей.

7.2.6.4. Учебный пример: использование сигналов в программе

fetchmail

Утилита

fetchmail обычно устанавливается для работы в качестве демона в фоновом режиме, который без вмешательства пользователя периодически собирает почту со всех удаленных узлов, указанных в конфигурационном файле, и отправляет ее локальному SMTP-слушателю на порт 25. Для того чтобы предотвратить постоянную загрузку сети,

fetchmail "засыпает" на определенное пользователем время (по умолчанию на 15 мин) между попытками сбора почты.

Команда fetchmail, введенная без аргументов, проверяет, присутствует ли в системе уже запущенный демон

fetchmail (проверка выполняется путем поиска PID-файла). Если это не так, то утилита

fetchmail запускается в обычном режиме, используя всю управляющую информацию, указанную в ее конфигурационном файле. С другой стороны, если демон уже запущен, то новый экземпляр

fetchmail только отправляет старому сигнал немедленно активизироваться и собрать почту, после чего новый экземпляр уничтожается. Команда fetchmail -q отправляет сигнал завершения всем запущенным демонам

fetchmail.

Таким образом, ввод команды fetchmail, в сущности, означает "немедленно опросить и оставить запущенным демон для последующего опроса; не выводить информацию о том, был ли демон уже запущен". Следует заметить, что подробности о том, какие именно сигналы использовались для активизации и завершения работы демона, представляют собой информацию, которую пользователю знать не обязательно.

7.2.6.5. Сокеты

Сокеты (sockets) были разработаны в BSD-ветви Unix как способ инкапсуляции доступа к сетям данных. Две программы, осуществляющие обмен данными через сокет, обычно используют двунаправленный поток байтов (существуют и другие режимы сокетов и методы передачи, но они имеют только второстепенное значение). Байтовый поток является как последовательным (т.е. все байты будут приняты в том же порядке, в котором они были отправлены), так и надежным (пользователи сокетов могут быть уверены, что базовая сеть в целях обеспечения гарантированной доставки осуществляет обнаружение ошибок и повтор передачи). Дескрипторы сокетов, полученные однажды, работают, по существу, подобно дескрипторам файлов.

Сокеты имеют одно важное отличие от операций чтения/записи. Если отправляемые байты поступают получателю, но принимающая машина не может отправить подтверждение ACK, то время ожидания TCP/IP-стека отправляющей машины истечет. Поэтому получение ошибки

не обязательно означает, что байты не поступили; возможно, получатель их использует. Данная проблема имеет значительные последствия для проектирования надежных протоколов, поскольку разработчик должны быть способен работать соответствующим образом, не зная, что было принято в прошлом. Ответы для локального ввода/вывода — "да" или "нет". Ответы для ввода/вывода сокета — "да", "нет", "возможно". И ничто не может гарантировать доставку — удаленная машина могла быть уничтожена кометой.Кен Арнольд.

Во время создания сокета задается

семейство протоколов , которое указывает сетевому уровню, как интерпретировать имя данного сокета. Сокеты обычно в связи с Internet рассматриваются как способ передачи данных между программами, запущенными на различных узлах. Таковым является семейство сокетов AF_INET, в котором адреса интерпретируются как пары "адрес узла-номер службы".

Однако семейство протоколов AF_UNIX (также известное как AF_LOCAL) поддерживает ту же абстракцию сокетов для обмена данными между двумя процессами на одной машине (имена интерпретируются как места расположения специальных файлов аналогично двунаправленным именованным каналам). Например, в клиентских программах и серверах, использующих систему X Window, для обмена данными обычно применяются сокеты AF_LOCAL.

Все современные Unix-системы поддерживают BSD-стиль сокетов, и в вопросе конструкции они обычно являются верным решением при использовании для двунаправленного IPC-взаимодействия не зависимо от того, где расположены взаимодействующие процессы. Желание повысить производительность может подтолкнуть разработчика к применению общей памяти, временных файлов или других технических приемов, которые вносят более строгие предположения о расположении, но в современных условиях лучше всего предполагать, что разрабатываемый код в будущем потребует расширения в целях поддержки распределенных операций. Еще более важно то, что данные локальные предположения могут означать, что внутренние блоки системы смешиваются больше, чем это должно быть в случае хорошей конструкции. Разделение адресных пространств, навязанное сокетами, является полезной особенностью, а не ошибкой.

Для того чтобы изящно, в духе Unix использовать сокеты, следует начинать с разработки используемого между ними протокола прикладного уровня, т.е. набора запросов и ответов, лаконично выражающего семантику данных, которыми будут обмениваться программы. Некоторые основные вопросы в проектировании протоколов прикладного уровня уже рассматривались в главе 5.

Сокеты поддерживаются во всех последних операционных системах семейства Unix, Microsoft Windows, а также в классической MacOS.

7.2.6.5.1. Учебный пример: PostgreSQL

PostgreSQL — программа с открытым исходным кодом для обработки баз данных. Если бы она была реализована как монолит, то это была бы одна программа с интерактивным интерфейсом, манипулирующая файлами баз данных непосредственно на диске. Интерфейс был бы неразрывно связанным с реализацией, и два экземпляра программы, пытающиеся одновременной манипулировать одной базой данных, создавали бы серьезные проблемы конфликтов и блокировок.

Вместо этого пакет PostgreSQL включает в себя сервер, который называется

postmaster, и по крайней мере 3 клиентских приложения. В машине в фоновом режиме запускается один серверный процесс

postmaster . Данный процесс имеет исключительный доступ к файлам базы данных. Он принимает запросы на мини-языке SQL-запросов посредством TCP/IP-сокетов, а также возвращает ответы в текстовом формате. Запущенный пользователем PostgreSQL-клиент открывает сеанс связи с сервером

postmaster и выполняет с ним SQL-транзакции. Одновременно сервер способен обрабатывать несколько клиентских сеансов, а их запросы последовательно располагаются так, что не пересекаются друг с другом.

Поскольку клиентская и серверная части обособлены, серверу не требуется выполнять

других задач, кроме интерпретации SQL-запросов от клиента и отправки SQL-отчетов в обратном направлении. С другой стороны, клиентам не требуется иметь какую-либо информацию о том, как хранится база данных. Клиенты могут быть специализированы для решения различных задач и иметь разные пользовательские интерфейсы.

Подобная организация весьма типична для баз данных в операционной системе Unix — настолько типична, что часто возможно сочетать и подбирать SQL-клиенты и SQL-серверы. Проблемы взаимодействия связаны с номером TCP/IP-порта SQL-сервера, а также с тем, поддерживают ли клиент и сервер один и тот же диалект SQL.

7.2.6.5.2. Учебный пример: Freeciv

В главе 6 игра Freeciv была представлена в качестве иллюстрации прозрачного формата данных. Однако более важным для поддержки игры с множеством участников является разделение кода на клиентскую и серверную части. Freeciv является характерным примером программы, в которой приложение должно быть распределено в сети и поддерживает связь через TCP/IP-сокеты.

Состояние запущенной игры Freeciv обслуживается серверным процессом, ядром игры. Игроки запускают GUI-клиенты, которые обмениваются информацией и командами с сервером посредством пакетного протокола. Вся игровая логика обрабатывается на сервере. Детали GUI-интерфейса обрабатываются клиентом. Различные клиенты поддерживают разные стили интерфейсов.

Организация данной игры весьма типична для сетевых игр с множеством участников. Пакетный протокол использует в качестве транспорта протокол TCP/IP, поэтому один сервер способен поддерживать клиентов, запущенных на различных узлах в Internet. В других играх, более похожих на симуляторы реального времени (особенно боевые игры от первого лица), используется протокол дейтаграмм Internet (UDP), и низкая задержка достигается за счет некоторой ненадежности доставки какого-либо определенного пакета. В таких играх пользователи, как правило, непрерывно выполняют управляющие действия, поэтому единичные потери пакетов допустимы, а задержка фатальна.

7.2.6.6. Общая память

Тогда как два процесса, использующие для информационного обмена сокеты, могут выполняться на различных машинах (и в действительности могут быть разделены Internet-соединением, "огибающим" половину планеты), общая память (shared memory) требует, чтобы поставщики и потребители данных одновременно находились в памяти одного компьютера. Однако, если процессы, обменивающиеся данными, могут получить доступ к одной физической памяти, то общая память будет самым быстрым способом передачи информации между ними.

Общая память может быть представлена различными API-интерфейсами, но в современных Unix-системах реализация обычно зависит от использования функции mmap

(2) для отображения файлов в общую память. В стандарте POSIX определяется средство

shm_open(3) с API-интерфейсом, поддерживающим использование файлов в качестве общей

памяти. Данная функция, главным образом, предоставляет операционной системе возможность не сбрасывать на диск данные псевдофайла.

Так как доступ к общей памяти автоматически не сериализуется в каком-либо порядке, подобном вызовам чтения и записи, программы с общей памятью должны обрабатывать проблемы конфликтов и взаимоблокировок самостоятельно, обычно путем использования семафоров, находящихся в совместно используемом сегменте. В данном случае возникают проблемы, подобные проблемам мультипроцессной обработки (их обсуждение приведено в конце данной главы), но они являются более управляемыми, так как по умолчанию программы

не используют общую память. Поэтому проблемы становятся более контролируемыми.

В системах, где это доступно и надежно работает, средство учета (scoreboard facility) Web-сервера Apache применяет общую память для обмена данными между главным процессом Apache и пулом распределения нагрузки образов Apache, которыми он управляет. В современных реализациях системы X также применяется общая память для передачи больших образов между клиентом и сервером, когда они находятся в памяти одной машины. В данном случае эта методика применяется для того, чтобы избежать издержек связи с использованием сокетов. Оба варианта применения представляют собой средство повышения производительности, обоснованное скорее опытом и тестами, чем архитектурным выбором.

Вызов

mmap(2) поддерживается во всех современных Unix-системах, включая Linux и версии BSD с открытым исходным кодом. Он описан в единой спецификации Unix (Single Unix Specification). Обычно он недоступен в Windows, классической MacOS и других операционных системах.

До того как появилась специализированная функция

mmap(2), общим способом сообщения двух процессов было открытие одного и того же файла и последующее удаление данного файла. Файл не удалялся до тех пор, пока не были закрыты все открытые дескрипторы данного файла, но некоторые старые Unix-системы использовали обнуление счетчика ссылок как указание на то, что обновление дисковой копии файла можно прекратить. Недостатком в этом случае было то, что вспомогательным запоминающим устройством была файловая система, а не устройство подкачки. Отключить файловую систему, на которой находился удаляемый файл, было невозможно до тех пор, пока не были закрыты использующие его программы, а подключение новых процессов к существующему сегменту общей памяти, выполненное таким способом, было в лучшем случае сложным.

После появления версии 7 и разделения ветвей BSD и System V эволюция межпроцессного взаимодействия в Unix стала развиваться в двух различных направлениях. Направление BSD привело к появлению сокетов. С другой стороны, ветвь AT&T развивала именованные каналы (как было сказано ранее) и IPC-средство, специально предназначенное для передачи двоичных данных и основанное на двунаправленных очередях сообщений в общей памяти. Это направление называется "System V IPC" или "Indian Hill IPC" (среди профессионалов прежней школы).

Верхний уровень System V IPC, уровень передачи сообщений, почти совершенно вышел из употребления. Нижний уровень, состоящий из общей памяти и семафоров, до сих пор находит широкое применение в условиях, когда необходимо выполнять блокировку с взаимным исключением и некоторое совместное использование глобальных данных между процессами, запущенными на одной машине. Данные средства общей памяти в System V развились в API с общей памятью стандарта POSIX, поддерживаемого в Linux, различных

версиях BSD, MacOS X и Windows, но не в классической MacOS.

Используя данные средства общей памяти и семафоров

(shmget(2), semget(2) и им подобные), можно избежать издержек копирования данных через сетевой стек. Данная техника интенсивно используется в крупных коммерческих базах данных (включая Oracle, DB2, Sybase и Informix).

7.3. Проблемы и методы, которых следует избегать

Несмотря на то, что BSD-сокеты через TCP/IP стали доминирующим IPC-методом в Unix, до сих пор продолжаются оживленные споры по поводу правильного способа разделения программ средствами мультипрограммирования. Некоторые устаревшие методы еще окончательно не умерли, а из других операционных систем заимствуются некоторые методики с сомнительной эффективностью (часто в связи с обработкой графики или программированием GUI-интерфейсов). Ниже рассматриваются некоторые небезопасные альтернативы.

7.3.1. Устаревшие IPC-методы в Unix

Операционная система Unix (созданная в 1969 году) предшествовала протоколу TCP/IP (появившемуся в 1980 году) и повсеместному распространению сетей позднее в 90-х годах прошлого века. Неименованные каналы, перенаправление и вызовы с созданием подоболочки были характерны для Unix с ранних дней ее существования, однако история Unix изобилует отжившими API-интерфейсами, связанными с устаревшими IPC-методами и сетевыми моделями, начиная со средства mx(), которое появилось в версии 6 (1976) и было отклонено до выхода версии 7 (1979).

В конечном итоге BSD-сокеты победили, когда IPC-механизм объединился с сетью. Однако это случилось только после 15 лет исследований, которые оставили за собой множество пережитков. Знать о них полезно, поскольку в Unix-документации, вероятно, найдутся ссылки на них, которые могут создать ошибочное мнение о том, что данные методы все еще используются. Более подробно эти устаревшие методы описываются в книге

"Unix Network Programming" [80].

Действительным оправданием для всех "мертвых" IPC-средств в старых Unix-системах ветви AT&T была политика. Группу поддержки Unix (The Unix Support Group) возглавлял менеджер низкого уровня, тогда как некоторыми проектами, в которых использовалась Unix, руководили вице-президенты. У них были возможности создавать непреодолимые запросы, и они не потерпели бы возражения, что большинство IPC-механизмов являются взаимозаменяемыми. Дуг Макилрой.

7.3.1.1. System V IPC

Средства System V IPC — средства передачи сообщений, основанные на имеющихся в System V возможностях общей памяти, которые были описаны ранее.

Программы, взаимодействующие с помощью System V IPC, обычно определяют общие протоколы, основанные на обмене короткими (до 8 Кб) двоичными сообщениями. Соответствующие справочные руководства доступны для

msgctl(2) и подобных функций. Поскольку данный стиль был почти полностью вытеснен текстовыми протоколами передачи данных между сокетами, примеры этой методики здесь не предоставляются.

Средства System V IPC присутствуют в Linux и других Unix-системах. Однако, поскольку они являются устаревшими, но используются не очень часто. Известно, что Linux-версия до середины 2003 года имела ошибки. Очевидно, никто не заботится об их исправлении.

7.3.1.2. Потоки

Потоки (streams) сетевого взаимодействия были разработаны Деннисом Ритчи для Unix Version 8 (1985). Их новая реализация называется STREAMS (именно так, в документации все буквы прописные). Впервые она стала доступной в версии 3.0 System V Unix (1986). Средство STREAMS обеспечивало дуплексный интерфейс (функционально не отличавшийся от BSD-сокетов, и подобно сокетам доступ к нему осуществлялся посредством обычных операций

read(2) и

write(2) после первоначальной установки) между пользовательским процессом и указанным драйвером устройства в ядре. Драйвер устройства мог быть аппаратным, таким как последовательная или сетевая плата, или исключительно программным псевдоустройством, установленным для передачи данных между пользовательскими процессами.

Интересной особенностью потоков и средства STREAMS[74] является то, что модули трансляции протокола можно внести в путь обработки ядра, так что данные устройства, используемого пользовательским процессом, проходя через дуплексный канал, фактически фильтруются. Данную возможность можно использовать, например, для реализации протокола построчного редактирования для терминального устройства. Также можно было бы реализовать такие протоколы, как IP или TCP, не встраивая их непосредственно в ядро.

Потоки стали попыткой упорядочить запутанную функцию ядра, которая называлась "протоколами линий" (line disciplines) — альтернативные режимы обработки символьных потоков, поступающих от последовательных терминалов и ранних локальных сетей. Однако по мере того как последовательные терминалы исчезали из вида, локальные сети Ethernet приобретали широкое распространение, а TCP/IP вытеснял другие наборы протоколов и мигрировал в ядра Unix, чрезвычайная гибкость, предоставляемая средством STREAMS, практически была утеряна. В 2003 году средство STREAMS поддерживалось в System V Unix, как и в некоторых гибридах System V/BSD, таких как Digital Unix и Solaris производства Sun Microsystems.

Linux и другие Unix-системы с открытыми исходными кодами фактически отказались от STREAMS. Модули ядра и библиотеки Linux доступны на сайте проекта LiS <http://www.gcom.com/home/linux/lis/>, но (к середине 2003 года) не интегрированы в основное ядро Linux. Они не поддерживаются в отличных от Unix операционных системах.

7.3.2. Методы удаленного вызова процедур

Несмотря немногочисленные исключения, такие как NFS (Network File System) и проект GNOME, попытки заимствовать технологии CORBA, ASN.1 и другие формы интерфейса удаленного вызова процедур в основном провалились. Данные технологии не прижились в Unix-культуре.

В основе этого, очевидно, лежит несколько проблем. Первая — RPC-интерфейсы не воспринимаются, т.е. опросить данные интерфейсы об их возможностях очень трудно. Кроме того, трудно осуществлять их мониторинг во время их работы без создания средств однократного применения настолько же сложных, насколько сложна программа, работа которой будет отслеживаться (некоторые из причин были рассмотрены в главе 6). Они имеют те же проблемы перекоса версий, что и библиотеки, но проблемы интерфейсов гораздо сложнее выявить, поскольку они распределены и неочевидны на этапе компоновки.

Существует еще одна проблема: интерфейсы, имеющие более развитые сигнатуры типов, также стремятся к большей сложности, а следовательно, являются более неустойчивыми. С течением времени происходит нарушение их онтологии, по мере того как ассортимент типов, проходящих через интерфейсы, неуклонно растет, а отдельные типы становятся более сложными. Нарушение онтологии становится проблемой, поскольку несогласованность структур более вероятна, чем несогласованность строк. Если онтология программ с каждой стороны не совпадает, то значительно затрудняется взаимодействие данных программы и устранение ошибок. Наиболее успешными RPC-приложениями (такими как Network File System) являются те, в которых прикладная область изначально имеет только несколько простых типов данных.

Обычным аргументом в пользу RPC является то, что данная технология допускает использование "более развитых" интерфейсов, чем методы, подобные текстовым потокам, — т.е. таких интерфейсов, в которых онтология типов данных является более сложной и специфичной для прикладной задачи. Однако нельзя забывать о правиле простоты. В главе 4 отмечалось, что одной из функций интерфейсов является создание заслонок, препятствующих взаимному проникновению деталей внутренней реализации модулей. Следовательно, главный аргумент в пользу RPC также является аргументом, подтверждающим возрастание глобальной сложности, вместо того, чтобы минимизировать ее.

RPC представляется методикой, которая поощряет создание крупных, причудливых, перепроектированных систем с неясными интерфейсами, высокой глобальной сложностью и серьезными проблемами надежности и перекоса версий — идеальный пример неуправляемых громоздких связующих уровней.

Технологии COM и DCOM в Windows являются, возможно, основными примерами того, насколько это может быть плохо, но существует множество других примеров. Компания Apple отказалась от технологии OpenDoc, а CORBA и однажды широко разрекламированная методика Java RMI исчезли с горизонта Unix, как только люди приобрели практический опыт работы с ними. Это вполне возможно, поскольку данные методы фактически решают не больше проблем, чем создают.

Эндрю С. Таненбаум (Andrew S. Tanenbaum) и Роберт ван Ренесс (Robbert van Renesse) подробно проанализировали общую проблему в статье

"A Critique of the Remote Procedure Call Paradigm" [83], которая должна послужить строгим

предостережением для тех, кто рассматривает возможность использования архитектуры, основанной на методиках RPC.

Все описанные проблемы могут обусловить долгосрочные трудности для сравнительно небольшого числа Unix-проектов, в которых используются RPC-методы. Среди таких проектов наиболее известным, вероятно, является GNOME[75]. Данные проблемы также вносят свой вклад в печально известные уязвимости незащищенных NFS-серверов.

С другой стороны, в мире Unix строго придерживаются прозрачных и воспринимаемых интерфейсов. Это одна из сил в основе непреходящей преданности Unix-культуры IPC-механизмам на основе текстовых протоколов. Часто утверждают, что издержки синтаксического анализа текстовых протоколов являются проблемой производительности по сравнению с двоичными RPC-протоколами, но RPC-интерфейсы склонны иметь проблемы задержек, которые представляются гораздо худшими, поскольку (a) невозможно без усилий заранее предсказать количество операций маршалинга и демаршалинга, которое будет задействовано определенным вызовом, и (b) RPC-модель поощряет программистов рассматривать сетевые транзакции как бесплатные. Добавление даже одного дополнительного обхода в интерфейс транзакции, как правило, добавляет сетевую задержку, достаточную для того, чтобы превысить любые издержки, связанные с синтаксическим анализом или маршалингом.

Даже если текстовые потоки были бы менее эффективными, чем RPC, потери производительности были бы незначительными и линейными, а такую проблему лучше решать с помощью модернизации аппаратного обеспечения, чем путем увеличения времени разработки или внесения дополнительной архитектурной сложности. Все потери производительности при использовании текстовых потоков, компенсируются благодаря возможности разрабатывать менее сложные системы, которые упрощают мониторинг, моделирование и понимание.

В настоящее время такие протоколы (XML-RPC и SOAP) являются интересным способом слияния RPC-методов и текстовых потоков в Unix. Протоколы XML-RPC и SOAP, будучи текстовыми и прозрачными, более приемлемы для Unix-программистов, чем небезопасные и тяжеловесные двоичные форматы сериализации, на смену которым они приходят. Несмотря на то, что они не решают всех глобальных проблем, указанных Танненбаумом и ван Ренессом, они действительно отчасти комбинируют преимущества текстовых потоков и RPC-методов.

7.3.3. Опасны ли параллельные процессы?

Хотя Unix-разработчики давно привыкли к вычислениям с помощью взаимодействующих процессов, среди них нет собственной традиции использования параллельных процессов (процессов, которые совместно используют все выделенное им адресное пространство). Параллельные процессы представляют собой недавнее заимствование извне, и тот факт, что Unix-программисты обычно испытывают к ним неприязнь, не является просто случайностью или исторически непредвиденным поворотом событий.

С точки зрения управления сложностью параллельные процессы являются плохой заменой легковесным процессам с собственными адресными пространствами. Идея параллельных процессов естественна для операционных систем с дорогим созданием подпроцессов и слабыми IPC-средствами.

По определению, несмотря на то, что дочерние параллельные процессы главного процесса

обычно обладают отдельными наборами локальных переменных, они совместно используют ту же глобальную память. Задача управления конфликтами и критическими областями в данном общем адресном пространстве является крайне сложным и богатым источником глобальной сложности и ошибок. Она может быть решена, однако по мере того, как растет сложность одного режима блокировки, соответственно растет вероятность конкуренции и взаимоблокировок благодаря непредвиденному взаимодействию.

Параллельные процессы являются источником ошибок, поскольку они могут чрезмерно просто получить слишком много сведений о внутренних состояниях друг друга. Не существует автоматической инкапсуляции, как это было бы между процессами с обособленными адресными пространствами, которые для обмена данными должны явно использовать IPC-методы. Таким образом, программы, разделенные на параллельные процессы, страдают не только от обычных проблем, связанных с конфликтами, но и от целых новых категорий ошибок, зависимых от синхронизации, которые крайне трудны даже для воспроизведения, не говоря об их устранении.

Разработчики параллельных процессов "сопротивляются" данной проблеме. Недавние реализации и стандарты демонстрируют возрастающей интерес к обеспечению локальной памяти процесса, которая предназначена для ограничения проблем, возникающих из-за совместного использования глобального адресного пространства. По мере того как API-интерфейсы двигаются в данном направлении, программирование с использованием параллельных процессов начинает все более походить на управляемое использование общей памяти.

Параллельные процессы часто препятствуют абстракции. В целях предотвращения взаимоблокировки часто требуется знать, используются ли в применяемой библиотеке параллельные процессы, чтобы избежать проблем взаимоблокировок, и если да, то как. Подобным образом на использование параллельных процессов в библиотеке могло бы воздействовать использование параллельных процессов на уровне приложения. Дэвид Корн.

В дополнение к вышесказанному, использование параллельной обработки приводит к снижению производительности, что, конечно же, умаляет ее преимущества по сравнению с традиционным разделением процессов. Хотя параллельная обработка может "избавиться" от некоторых издержек быстрого переключения контекста процессов, блокировка общих структур данных с целью предотвратить пересечение параллельных процессов, может быть такой же дорогостоящей.

X-сервер, способный выполнять буквально миллионы операций в секунду,

не разделяется на параллельные процессы. В нем используется цикл poll/select. Многочисленные усилия создать мультипроцессную реализацию привели к негативным результатам. Цена блокировки и разблокировки становится слишком высокой для систем настолько же чувствительных к производительности, насколько чувствительны к ней графические серверы.Джим Геттис.

Данная проблема является фундаментальной и продолжает оставаться спорной в проектировании ядер Unix для симметричной многопроцессорной обработки. По мере того как блокировка ресурсов становится все более точной, задержка ввиду издержек блокировки может довольно быстро превысить преимущества от блокировки меньшего объема оперативной памяти.

Одна из трудностей, связанных с параллельными процессами, заключается в том, что стандарты данной технологии до середины 2003 года оставались слабыми и недоопределенными. Теоретически согласующиеся библиотеки для таких Unix-стандартов, как POSIX (1003.1c) могут, тем не менее, проявить опасные различия при функционировании

на различных платформах, особенно в аспекте сигналов, взаимодействия с остальными методами IPC и времени освобождения ресурсов. Операционные системы Windows и классическая MacOS обладают собственными моделями параллельной обработки и средствами прерывания, которые очень отличаются от имеющихся в Unix и часто требуют значительных усилий при переносе даже простых конструкций. Вследствие этого рассчитывать на переносимость программ, использующих параллельные процессы, не приходится.

Более широкое обсуждение данного вопроса представлено в статье

"Why Threads Are a Bad Idea" [59].

7.4. Разделение процессов на уровне проектирования

Рассмотрим конкретные рекомендации по использованию описанных выше методов.

Прежде всего, необходимо отметить, что временные файлы, более интерактивный способ связи главного/подчиненного процессов, сокеты, RPC и все остальные методы двунаправленного межпроцессного взаимодействия являются эквивалентными на некотором уровне — все они представляют собой только способы обмена данными во время работы программ. Большую часть из того, что делается сложным способом с использованием сокетов или общей памяти, можно было бы реализовать примитивным путем, используя временные файлы в качестве почтовых ящиков и сигналы для доставки уведомлений. Отличия сосредоточены на границах и состоят в том, как программы устанавливают соединения, где и когда выполняется маршалинг и демаршалинг сообщений, какого рода проблемы буферизации могут возникнуть и каковы элементарные гарантии, получаемые с помощью сообщений (т.е. до какой степени можно быть уверенным, что результат одной операции отправки с одной стороны отразится в виде одного события получения на другой стороне).

При рассмотрении PostgreSQL было отмечено, что эффективным способом сдерживания сложности является разделение приложения на пару "клиент-сервер". Клиент и сервер PostgreSQL сообщаются посредством протокола прикладного уровня через сокеты, однако в модели проектирования изменилось бы очень немногое, если бы они использовали любой другой двунаправленный IPC-метод.

Данный вид разделения является особенно эффективным в ситуациях, когда множество экземпляров приложения должны управлять доступом к ресурсам, которые используются ими совместно. Управлять всем распределением ресурсов может один серверный процесс, или каждый из взаимодействующих равноправных процессов может принимать ответственность за некоторый критический ресурс.

Клиент-серверное разделение может также способствовать распределению приложений, интенсивно использующих ресурсы процессора, среди множества узлов. Или такое разделение может сделать их пригодными для распределенных вычислений через Internet (как в случае с игрой Freeciv). Связанная с этим модель

CLI-сервера рассматривается в главе 11.

Поскольку все эти равноправные IPC-методики являются на некотором уровне идентичными, следует оценивать их главным образом по величине издержек программной сложности, которые они создают, а также по степени непрозрачности, вносимой ими в разрабатываемые

конструкции. Именно поэтому в конечном итоге BSD-сокеты опередили остальные IPC-методы Unix, а RPC-методы не смогли привлечь к себе значительный интерес.

Параллельные процессы имеют фундаментальные отличия. Вместо поддержки взаимодействия между различными программами, они поддерживают некий вид разделения времени внутри экземпляра одной программы. Вместо того чтобы быть способом разделения большой программы на мелкие с более простым поведением, параллельная обработка является строго средством повышения производительности, и поэтому имеет все связанные с этим проблемы, а также ряд собственных.

Соответственно, несмотря на то, что проектировщикам следует искать пути разделения крупных программ на более простые взаимодействующие процессы, использование параллельной обработки внутри процессов должно быть скорее последним средством, чем первым. Часто обнаруживается, что их использования можно избежать. Если вместо параллельных процессов можно использовать ограниченную общую память и семафоры, асинхронные I/O-операции с использованием SIGIO или цикл

poll(2)/select(2) , то такой возможностью следует воспользоваться. Поддерживайте простоту; используйте менее сложные методики.

Комбинация параллельных процессов, интерфейсов удаленного вызова процедур и тяжеловесного объектно-ориентированного дизайна является особенно опасной. Расчетливое и изящное применение любого из этих технических приемов может оказаться весьма полезным, но от проектов, где предполагается использовать все три, следует решительно отказаться.

Ранее уже отмечалось, что программирование в реальном мире направлено на управление сложностью и инструменты для управления сложностью очень полезны. Но когда эффект от их применения заключается в распространении сложности, а не в управлении ею, то лучше будет отказаться от них и начать проект с нуля. Никогда не забывайте об этом.

8

Мини-языки: поиск выразительной нотации

Хорошая нотация обладает тонкостью и выразительностью, которая со временем делает ее почти похожей на живого учителя.

The World of Mathematics (1956) —Бертранд Рассел (Bertrand Russell)

Одним из самых последовательных результатов крупномасштабных исследований ошибок в программировании является то, что уровень ошибок программиста, выраженный в количестве дефектов на 100 строк кода, почти не зависит от языка, на котором написана программа[76]. Высокоуровневые языки, которые позволяют добиться больших результатов, используя меньшее количество строк, также означают меньшее количество ошибок.

В Unix имеется давняя традиция поддержки небольших языков, предназначенных для определенной прикладной области, языков, которые могут способствовать в радикальном сокращении количества строк кода в программах. Примеры узкоспециальных (domain-specific) языков включают в себя многочисленные языки разметки текстов (

troff, eqn, tbl, pic, grap), утилиты оболочки (

awk, sed, dc, bc) и средства разработки программного обеспечения

(make, yacc, lex). Невозможно провести четкие границы между узкоспециальными языками и более гибким видом файлов конфигурации программ (

sendmail, BIND, X), или форматами файлов данных, или языками сценариев (которые рассматриваются в главе 14).

В сообществе Unix для таких языков узкоспециального назначения исторически определилось название "малые языки" или "мини-языки" , поскольку ранние их примеры были небольшими и имели небольшую сложность по сравнению с универсальными языками (в настоящее время широко используются все 3 термина для данной категории). Однако если предметная область сложна (тем, что имеет множество различных примитивных операций или включает в себя манипуляцию сложными структурами данных), то для нее может понадобиться прикладной язык, гораздо более сложный, чем некоторые универсальные языки. В данной книге используется традиционный термин "мини-язык" (minilanguage), для того чтобы подчеркнуть, что мудрое решение обычно заключается в сохранении данных конструкций небольшими и простыми насколько это возможно.

Узкоспециальный небольшой язык — чрезвычайно мощная конструкторская идея. Он позволяет определить собственный высокоуровневый язык для указания соответствующих методов, правил и алгоритмов, направленных на разрешение ближайшей задачи, сокращая глобальную сложность по сравнению с конструкцией, в которой для тех же целей используется жестко встроенный низкоуровневый код. Прийти к использованию мини-языка можно как минимум тремя путями, два из которых хороши, а один опасен.

Один из верных путей заключается в том, чтобы заранее осознать возможность использования конструкции на основе мини-языка, для того чтобы поднять данную спецификацию проблемы программирования на уровень выше к форме записи, которая является более компактной и выразительной, чем нотация, поддерживаемая в универсальном языке. Как и в случае с генерацией кода и создания программ, управляемых данными, мини-язык позволяет извлечь практическое преимущество из того факта, что количество ошибок в программном обеспечении будет почти не зависеть от уровня используемого языка; использование более выразительных языков означает более короткие программы и меньшее количество ошибок.

Второй правильный путь — заметить, что один из файловых форматов разрабатываемой спецификации очень похож на мини-язык, т.е. в нем развиваются сложные структуры и подразумеваются действия в контролируемом приложении. Можно ли с помощью данного языка попытаться описать управляющую логику так же, как форматы данных? Если это так, то, возможно, настало время перевести управляющую логику из неявного вида в явный в языке спецификации.

Ошибочный путь к конструкции мини-языка — это растягивать путь к нему, постепенно добавляя заплатки и сложные функции. На этом пути файл спецификации содержит задатки более скрытой управляющей логики и более замысловатых специализированных структур до тех пор, пока незаметно не станет сложным уникальным языком. Несколько "легендарных кошмаров встают" на этом пути. Каждый Unix-гуру вздрогнет при упоминании конфигурационного файла sendmail.cf, связанного с почтовым транспортом

sendmail.

К сожалению, большинство разработчиков создают свой первый мини-язык ошибочным способом и только позднее осознают, насколько он запутан. Как очистить мини-язык? Иногда

ответ предполагает переосмысление конструкции всего приложения. Другим печально известным примером был редактор TECO, в котором возник первый макрос, а затем появились циклы и условные операторы по мере того, как программисты хотели использовать его для упаковки редактирующих подпрограмм с возрастающей сложностью. Созданная в результате уродливая конструкция была в конечном итоге исправлена путем переработки всего редактора, основанного на заранее продуманном языке. Так развивался Emacs Lisp (который рассматривается ниже).

Все достаточно сложные файлы спецификаций поднимаются до уровня мини- языков. Поэтому часто единственный способ обезопасить себя от создания плохого мини-языка заключается в том, чтобы знать, как создать хороший мини-язык. Это не должно быть сопряжено с неимоверными трудностями и наличием особых знаний относительно формальной теории языков. Вполне достаточно практического проектирования с помощью современных инструментов, изучения немногих относительно простых технических приемов и ознакомления с хорошими примерами.

В данной главе рассматриваются все виды мини-языков, обычно поддерживаемых в Unix. Кроме того, ниже определяются ситуации, в которых каждый из них представляет эффективное конструктивное решение. При этом данная глава не является исчерпывающим каталогом Unix-языков, а скорее направлена на выявления принципов конструирования, задействованных в структурировании приложений вокруг мини-языка. Универсальные языки программирования более подробно рассматриваются в главе 14.

Начать следует с небольшой классификации, которая поможет лучше понять дальнейший материал.

8.1. Классификация языков

Все языки, представленные на рис. 8.1, описываются в учебных примерах этой или других глав данной книги. Описание универсальных интерпретаторов, показанных в правой части схемы, приведено в главе 14.

В главе 5 рассматривались Unix-соглашения для файлов данных. В них имеется определенный спектр сложности. На самом низком уровне находятся файлы, в которых создаются простые ассоциации между именами и свойствами, хорошими примерами таких форматов являются файлы /etc/passwd и .newsrc. Далее представлены форматы, которые осуществляют маршалинг или сериализацию структур данных. Одинаково хорошими примерами в данном случае являются форматы PNG и SNG.

Структурированные форматы файлов данных начинаются на границе мини-языков, когда они выражают не только структуру, но и действия, выполняемые в некоторой интерпретирующей среде (т.е. памяти за пределами самого файла данных). XML-разметка стремится "перешагнуть" эту границу. Примером такого мини-языка, представленным в данной главе, является

Glade , генератор кода для создания GUI-интерфейсов. Форматы, которые одновременно разработаны для чтения и записи человеком (скорее человеком, чем программами) и используются для генерации кода, прочно укрепились в области мини-языков. Классическими примерами являются утилиты

yacc и

lex . Программы

glade, yacc и

lex описываются в главе 9.

Макропроцессор Unix,

m4 представляет собой другой очень простой декларативный мини-язык (т.е. язык, в котором программа выражается как набор желаемых связей или ограничений, а не как явные действия). Он часто используется в качестве препроцессора для других мини-языков.

Рис. 8.1. Классификация языков

make-файлы Unix, предназначенные для автоматизации процесса сборки, выражают зависимости между исходными и производными файлами[77], а также команды, необходимые для создания каждого производного файла из его исходного кода. При выполнении команда

make использует данные объявления для обхода предполагаемого дерева зависимостей, выполняя наименьшую необходимую работу для обновления сборки. Подобно спецификациям

yacc и

lex, make-файлы являются декларативным мини-языком. Они устанавливают ограничения, которые предполагают действия, выполняемые в интерпретирующей среде (в данном случае в той части файловой системы, где расположены исходные и сгенерированные файлы), make-файлы дополнительно рассматриваются в главе 15.

Язык XSLT, который используется для описания трансформаций XML-файлов, соответствует верхнему уровню сложности декларативных мини-языков. Он довольно сложен для того, чтобы рассматривать его как мини-язык, однако разделяет некоторые важные характеристики таких языков, которые подробнее рассматриваются ниже при изучении XSLT.

Спектр мини-языков простирается от декларативных (с неявными действиями) к императивным (с явными действиями). Синтаксис файла конфигурации программы

fetchmail(1) можно рассматривать либо как очень слабый императивный язык, либо как декларативный язык с неявной управляющей логикой. Языки обработки текстов troff и PostScript являются императивными языками с большим количеством встроенной специальной информации о прикладной области.

Некоторые императивные мини-языки для решения специальных задач граничат с универсальными интерпретаторами. Они достигают данного уровня, когда явно являются

языками Тьюринга , т.е. они могут выполнять условные операции и циклы (или рекурсию)[78] с функциями, которые предназначены для использования в качестве управляющих структур. В отличие от них, некоторые языки только отчасти являются языками Тьюринга. В них имеются функции, которые можно использовать для реализации управляющих структур как побочный эффект того, для чего они фактически предназначены.

Интерпретаторы

bc(1) и

dc(1), рассмотренные в главе 7, являются хорошими примерами специализированных императивных мини-языков, которые явно являются языками Тьюринга.

Такие языки, как Emacs Lisp и JavaScript, находятся в области универсальных интерпретаторов. Языки Emacs Lisp и JavaScript предназначены для использования в качестве полных языков программирования, работающих в специализированных средах. Более подробно они описываются ниже при рассмотрении встроенных языков сценариев.

Область интерпретаторов представляет собой область возрастающей неопределенности. Оборотной стороной этого является то, что более универсальный интерпретатор включает в себя меньше предположений о среде, в которой он работает. С возрастающей неопределенностью обычно приходит более развитая онтология типов данных. Shell и Tcl обладают сравнительно простой онтологией, а Perl, Python и Java — более сложной. Данные универсальные языки подробнее рассматриваются в главе 14.

8.2. Применение мини-языков

Разработка программ с помощью мини-языков затрагивает две отдельные проблемы. Одна из них заключается в том, чтобы уметь пользоваться имеющимися в инструментарии мини-языками и понимать, когда их можно применять такими, как они есть. Другая проблема — знать, когда целесообразно разрабатывать для приложения нестандартный мини-язык. Для того чтобы помочь читателю развить оба аспекта конструкторского мышления, почти половина данной главы состоит из учебных примеров.

8.2.1. Учебный пример:

sng

В главе 6 рассматривалась утилита

sng(1), преобразовывающая PNG-файл в редактируемую полностью текстовую форму. Формат файлов данных SNG заслуживает повторного рассмотрения здесь для контраста, поскольку он не вполне является узкоспециальным мини-языком. Он описывает расположение данных, но не связывает с ними какую-либо предполагаемую последовательность действий.

Однако SNG действительно имеет одну общую важную характеристику с узкоспециальными мини-языками, которую не поддерживают структурированные двоичные форматы данных, подобные PNG, — прозрачность. Структурированные файлы данных позволяют без использования мини-языка взаимодействовать средствам редактирования, преобразования и создания, которые не имеют информации о конструкторских "предположениях" друг друга. В случае SNG добавляется то, что данный формат как узкоспециальный мини-язык, предназначен для простого просмотра и редактирования с помощью универсальных средств.

8.2.2. Учебный пример: регулярные выражения

Одним из видов спецификации, который периодически появляется в инструментах Unix и языках сценариев, является

регулярное выражение (regular expression, или regexp для краткости). Здесь регулярные выражения рассматриваются как декларативный мини-язык для описания текстовых шаблонов. Часто регулярные выражения встраиваются в другие мини-языки. Регулярные выражения настолько распространены, что их едва ли можно считать мини-языком, однако они заменяют то, что в противном случае представляло было собой огромные объемы кода, реализующего различные (и несовместимые) возможности поиска.

В данном введении не рассматриваются такие подробности, как POSIX-расширения, обратные ссылки и особенности интернационализации. Более подробное изложение способа их применения представлено в книге

"Mastering Regular Expressions" [22].

Регулярные выражения описывают шаблоны, которые могут либо совпадать, либо не совпадать со строками. Простейшим средством для работы с регулярными выражениями является утилита

grep(1) , фильтр, который переправляет со стандартного ввода на стандартный вывод каждую строку, соответствующую указанному регулярному выражению. Форма записи регулярных выражений кратко представлена в таблице 8.1.

Таблица 8.1. Примеры регулярных выражений Регулярное выражение Соответствующая строка "x.y" x, за которым следует любой символ с последующим y "x\.y" x, за которым следует точка с последующим y "xz?y" x, за которым следует не более одного символа z с последующим y, т.е. "xy" или "xzy", но не "xz" или "xdy" "xz*y" x, за которым следует любое количество символов z, за которыми следует y, т.е. "xy" или "xzy" или "xzzzy", но не "xz" или "xdy" "xz+y" x, за которым следует один или несколько экземпляров символа z, за которыми следует y, т.е. "xzy" или "xzzy", но не "xy", "xz" или "xdy" "s[xyz]t" s, за которым следует любой из символов x, y или z, за которым следует t, т.е. "sxt", "syt" или "szt", но не "st" или "sat" "a[x0-9]b" a, за которым следует либо x, либо символ в диапазоне 0-9, за которым следует b, то есть, "axb", "a0b" или "a4b", но не "ab" или "aab" "s[^xyz] t" s, за которым следует любой символ, кроме x, y или z, за которым следует t, т.е. "sdt" или "set", но не "sxt", "syt" или "szt" "s[^x0-9]t" s, за которым следует любой символ, кроме x или символа в диапазоне 0-9, за которым следует t, т.е. "slt" или "smt", но не "sxt", "s0t" или "s4t" "^x” x в начале строки, т.е. "xzy" или "xzzy", но не "yzy" или "yxy" "x$" x в конце строки, т.е. "yzx" или "yx", но не "yxz" или "zxy"

Существует большое количество второстепенных вариантов записи регулярных выражений.

1.

Выражения-маски. Ограниченный набор соглашений по применению символов-шаблонов (wildcard), использовавшийся в ранних оболочках Unix для сопоставления имен файлов. Существует всего 3 символа-шаблона: * — соответствует любой последовательности символов (как .* в других вариантах); ? — соответствует любому единичному символу (как . в других вариантах); [...] — соответствует классу символов как в других вариантах. В некоторых оболочках (

csh, bash, zsh) позднее был добавлен шаблон {} для выбора подстроки. Таким образом, выражение x{a,b}c соответствует строкам xac или xbc, но не xc. В некоторых оболочках выражения-маски получили дальнейшее развитие в направлении расширения регулярных выражений.

2.

Базовые регулярные выражения. Форма записи, принятая в исходной утилите

grep(1) для извлечения из файла строк, соответствующих заданному регулярному выражению. Выражения этого типа также применяются в строковом редакторе

ed(1) и потоковом редакторе

sed(1) . Профессионалы старой школы Unix считают данное выражение основной, или "унифицированной", разновидностью регулярных выражений. Пользователи, впервые столкнувшиеся с более современными инструментами, склонны использовать расширенную форму, которая описана ниже.

3.

Расширенные регулярные выражения. Запись, принятая в расширенной версии grep,

egrep(1) для извлечения из файла строк, соответствующих заданному регулярному выражению. Регулярные выражения в Lex и редакторе

Emacs весьма близки к egrep-разновидности.

4.

Регулярные выражения языка Perl. Форма записи, принятая в regexp-функциях языков Perl и Python. Выражения этого типа являются более мощными по сравнению с egrep-вариантом.

После рассмотрения основных примеров в таблице 8.2 приведена сводка стандартных шаблонов для регулярных выражений. Следует отметить, что в таблицу не включен вариант выражений-масок, поэтому запись "для всех" означает только 3 типа: базовый, расширенный/Emacs и Perl/Python[79].

Таблица 8.2. Введение в операции с регулярными выражениями Символ-шаблон Поддерживается Соответствующая строка \ во всех Начало escape-последовательности. Определяет, следует ли интерпретировать последующий знак как шаблон. Последующие буквы или цифры интерпретируются различными способами в зависимости от программы . во всех Любой символ ^ во всех Начало строки $ во всех Конец строки [...] во всех Любой из символов, указанных в скобках [^...] во всех Любые символы,

кроме указанных в скобках * во всех Любое количество экземпляров предыдущего элемента ? egrep/Emacs, Perl/Python Ни одного или один экземпляр предыдущего элемента + egrep/Emacs, Perl/Python Один или несколько экземпляров предыдущего элемента {n} egrep, Perl/Python; как \{n\} в Emacs В точности n повторений предыдущего элемента. Не поддерживается некоторыми старыми regexp-средствами {n,} egrep, Perl/Python; как \{n,\} в Emacs n или более повторений предыдущего элемента. Не поддерживается некоторыми старыми regexp-средствами {m,n} egrep, Perl/Python; как \{m,n\} в Emacs Минимум m и максимум n повторений предыдущего элемента. Не поддерживается некоторыми старыми regexp-средствами | egrep, Perl/Python; как \| в Emacs Элемент слева или справа. Обычно используется с некоторой формой группирующих разделителей (...) Perl/Python; как \(...\) в более старых версиях Интерпретировать данный шаблон как группу (в более новых regexp-функциях, например в языках Perl и Python). Более старые средства, такие как regexp-функции в Emacs и в утилите grep требуют записи \(...\)

В новых языках с поддержкой регулярных выражений установилась практика

Perl/Python-варианта. Он является более прозрачным, чем остальные, особенно потому, что обратная косая черта перед не алфавитно-цифровым символом всегда означает, что данный символ трактуется буквально, что значительно устраняет путаницу при ссылке на элементы регулярных выражений.

Регулярные выражения являются исключительным примером того, насколько лаконичным может быть мини-язык. Простые регулярные выражения отражают режим распознавания, который иначе пришлось бы реализовывать с помощью сотен строк туманного, чреватого ошибками кода.

8.2.3. Учебный пример: Glade

Glade представляет собой средство разработки интерфейсов для библиотеки X-инструментария[80] GTK с открытым исходным кодом.

Glade позволяет разрабатывать GUI-интерфейс путем интерактивного выбора, размещения и модификации элементов управления на панели интерфейса. GUI-редактор создает XML-файл, описывающий проектируемый интерфейс. Полученный файл, в свою очередь, можно передать одному из нескольких генераторов кода, которые непосредственно создают C, C++, Python- или Perl-код для интерфейса. Сгенерированный код затем вызывает создаваемые разработчиком функции, определяющие поведение интерфейса.

XML-формат

Glade для описания GUI-интерфейсов является хорошим примером простого узкоспециального мини-языка. В примере 8.1 показан Glade-формат для GUI-интерфейса "Hello, world!".

Адекватная спецификация в Glade-формате предполагает набор действий, предпринимаемых GUI-интерфейсом в ответ на действия пользователя. GUI-интерфейс

Glade интерпретирует данные спецификации как структурированные файлы данных. С другой стороны, генераторы кода

Glade используют их для написания программ, реализующих GUI. Для некоторых языков (включая Python) существуют библиотеки времени выполнения, которые позволяют пропустить этап генерирования кода и просто создавать GUI непосредственно во время обработки XML-спецификации (интерпретируя

Glade- разметку вместо того, чтобы компилировать ее). Таким образом, разработчик получает выбор: эффективное использование пространства ценой скорости запуска или наоборот.

Программисту, освоившему подробности XML-формата, разметка

Glade представляется довольно простым языком. Она решает только две задачи: объявляет иерархии GUI-элементов и связывает свойства с элементами управления. Чтобы прочесть спецификацию, представленную в примере, разработчику фактически не требуется досконально разбираться в работе

glade . Действительно, имея опыт программирования с помощью GUI-инструментариев и читая данную спецификацию, можно сразу довольно наглядно представить себе интерфейс, создаваемый

glade на ее основе. (Для тех, кто не имеет такого опыта, можно отметить, что данная спецификация позволяет получить однокнопочный элемент управления в окне).Пример 8.1. Glade-спецификация "Hello, world!"

```
&lt;?xml version="1.0"?&gt;
&lt;GTK-Interface&gt;
 &lt;widget&gt;
  &lt;class&gt;GtkWindow&lt;/class&gt;
  &lt;name&gt;HelloWindow&lt;/name&gt;
  &lt;border_width&gt;5&lt;/border_width&gt;
  &lt;Signal&gt;
   &lt;name&gt;destroy&lt;/name&gt;
   &lt;handler&gt;gtk_main_quit&lt;/handler&gt;
  &lt;/Signal&gt;
  &lt;title&gt;Hello&lt;/title&gt;
  &lt;type&gt;GTK_WINDOW_TOPLEVEL&lt;/type&gt;
  &lt;position&gt;GTK_WIN_POS_NONE&lt;/position&gt;
  &lt;allow_shrink&gt;True&lt;/allow_shrink&gt;
  &lt;allow_grow&gt;True&lt;/allow_grow&gt;
  &lt;auto_shrink&gt;False&lt;/auto_shrink&gt;
  &lt;widget&gt;
   &lt;class&gt;GtkButton&lt;/class&gt;
   &lt;name&gt;Hello World&lt;/name&gt;
   &lt;can_focus&gt;True&lt;/can_focus&gt;
   &lt;Signal&gt;
    &lt;name&gt;clicked&lt;/name&gt;
    &lt;handler&gt;gtk_widget_destroy&lt;/handler&gt;
    &lt;object&gt;HelloWindow&lt;/object&gt;
   &lt;/Signal&gt;
   &lt;label&gt;Hello World&lt;/label&gt;
  &lt;/widget&gt;
 &lt;/widget&gt;
```

</GTK-Interface>

Такая прозрачность и простота являются следствием хорошей конструкции мини- языка. Соответствие между нотацией и объектами предметной области весьма очевидно. Связи между объектами выражаются непосредственно, а не через именованные ссылки или другое непрямое преобразование, которого необходимо придерживаться.

Основой функциональный тест для подобного мини-языка прост: возможно ли разобраться в данном языке, не изучая руководство? Для значительного количества случаев использования

Glade это так. Например, зная константы C-уровня, которые в GTK используются для описания параметров позиционирования окна, можно распознать константу GTK_WIN_POS_NONE как одну из них и немедленно получить возможность изменить параметры позиционирования, связанные с данным GUI-интерфейсом.

Преимущества использования

Glade должны быть очевидны. Данная программа специализируется на создании кода, что освобождает разработчика от необходимости его собственной специализации. То есть

Glade принимает на себя одну рутинную задачу, которую в противном случае придется решать вручную. Кроме того, разработчик избавляется от одного из источников ошибок, неизбежных при ручном кодировании.

Более подробная информация, включая исходный код, документацию и ссылки на примеры приложений, доступны на странице проекта

Glade <http://glade.gnome.org/>. Программа

Glade перенесена на платформу Microsoft Windows.

8.2.4. Учебный пример:

m4

Макропроцессор

m4(1) интерпретирует декларативный мини-язык для описания трансформаций текста. Программа

m4 представляет собой набор макросов, который определяет способы преобразования одних текстовых строк в другие. В результате применения описаний к входному тексту с командами

m4 происходит макрорасширение и на выходе создается текст. (Препроцессор C предоставляет аналогичные службы для компиляторов C, хотя и в несколько отличающемся стиле.)

В примере 8.2 показана макрокоманда

m4 , которая заставляет утилиту

m4 преобразовывать каждое вхождение строки "OS" в тесте ввода в строку "operating system" на выводе. Данный пример тривиален.

m4 поддерживает макросы с аргументами, которые можно использовать для более сложных операций, чем просто преобразование одной фиксированной строки в другую. Ввод info m4 в командную строку оболочки позволит получить справочную документацию по данному языку.Пример 8.2. Макрокоманда

m4

define(`OS', `operating system')

Макроязык

m4 поддерживает условные операторы и рекурсию, комбинацию которых можно использовать для реализации циклов, что и было задумано разработчиками. Макроязык

m4 преднамеренно создан как язык Тьюринга. Однако было бы глубоким заблуждением пытаться использовать его в качестве универсального языка.

Макропроцессор

m4 обычно используется как препроцессор для мини-языков, которые испытывают недостаток встроенного понятия именованных процедур или встроенной функции включения файлов. Это простой путь для расширения синтаксиса базового языка, так чтобы комбинация с

m4 поддерживала обе эти функции.

Одним из широко известных способов применения

m4 является очистка (или, по крайней мере, эффективное сокрытие) другой конструкции мини-языка, который ранее в данной главе был назван плохим примером. В настоящее время большинство системных администраторов генерируют конфигурационные файлы sendmail.cf с помощью пакета макрокоманд

m4, который поставляется с дистрибутивом

sendmail . Макросы начинают с имен функций (или пар имя/значение) и генерируют соответствующие (гораздо более уродливые) строки на языке конфигурации программы

sendmail .

Однако

m4 следует использовать осмотрительно. Опыт Unix научил разработчиков мини-языков остерегаться макрорасширения. Причины этого описываются далее в настоящей главе.

8.2.5. Учебный пример: XSLT

Язык XSLT, подобно макросам

m4 , является языком для описания трансформаций текстового потока. Однако он делает гораздо больше, чем просто подмену макрокоманд. Он описывает трансформации XML-данных, включая создание запросов и отчетов. XSLT — язык для написания таблиц стилей XML. В целях практического применения рекомендуется изучить описание обработки XML-документов в главе 18. XSLT описан стандартом Консорциума World Wide Web и имеет несколько реализаций с открытым исходным кодом.

m4 являются как исключительно декларативными языками, так и языками Тьюринга, но XSLT поддерживает только рекурсии и не поддерживает циклы. Он весьма сложен, несомненно, наиболее трудный для освоения язык из всех упомянутых в учебных примерах данной главы, а возможно, и самый трудный язык из упомянутых в этой книге[81].

Несмотря на сложность, XSLT действительно является мини-языком. Он обладает важными (хотя и не всеми) характеристиками своего класса:

• ограниченная онтология типов, в частности, без каких-либо аналогов структур записи или массивов;

• ограниченный интерфейс для связи с внешним миром; XSLT-процессоры предназначены для фильтрации стандартного ввода на стандартный вывод, с ограниченными возможностями считывать и записывать файлы; они не способны открывать сокеты или запускать подкоманды.

Программа в примере 8.3, трансформирует XML-документ так, что каждый атрибут каждого элемента трансформируется в новую пару тегов, непосредственно заключенную внутри элемента. Значение атрибута представлено как содержимое пары тегов.

Краткий обзор XSLT приведен здесь отчасти для иллюстрации того факта, что понятие "декларативности" не подразумевает "простоту" или "слабость", и главным образом ввиду того, что необходимость работы с XML-документами рано или поздно приводит к проблеме, каковой является XSLT.

Книга

"XSLT: Mastering XML Transformations" [84] является хорошим введением в изучение данного языка. Краткие учебные материалы с примерами доступны в Web[82].Пример 8.3. XSLT-программа

```
&lt;?xml version="1.0"?&gt;

&lt;xsl:stylesheet

 xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"&gt;

 &lt;xsl:output method="xml"/&gt;

 &lt;xsl:template match="*"&gt;

  &lt;xsl:element name="{name()}"&gt;

   &lt;xsl:for-each select="@*"&gt;

    &lt;xsl:element name="{name()}"&gt;

     &lt;xsl:value-of select="."/&gt;

    &lt;/xsl:element&gt;

   &lt;/xsl:for-each&gt;

   &lt;xsl:apply-templates select="*|text()"/&gt;

  &lt;/xsl:element&gt;
```

</xsl:template>

</xsl:stylesheet>

8.2.6. Учебный пример: инструментарий Documenter's Workbench

Программа

troff(1) , средство форматирования текстов, была, как отмечалось в главе 2, первоначальным главным приложением операционной системы Unix. Программа

troff является наиболее важной в наборе форматирующих средств (получивших коллективное название DWB, (Documenter's Workbench — автоматизированное рабочее место документатора), каждое из которых является отдельным узкоспециальным мини-языком. Большинство из них являются либо препроцессорами, либо постпроцессорами для troff-разметки. В Unix-системах с открытыми исходными кодами используется расширенная реализация DWB (которая называется

groff(1)), созданная Фондом свободного программного обеспечения.

Подробнее программа

troff рассматривается в главе 18. Здесь достаточно отметить, что она представляет собой хороший пример императивного мини-языка, граничащего с полностью проработанным интерпретатором (в

troff поддерживаются условные операции и рекурсия, но нет циклов;

troff — отчасти язык Тьюринга).

Постпроцессоры ("драйверы" в терминологии DWB) обычно невидимы для пользователей

troff . Первоначально созданные troff-коды для отдельных наборных машин были доступны группе разработки Unix в 1970 году. Позднее они были улучшены до аппаратно-независимого мини-языка для размещения текста и простой графики на страницах. Постпроцессоры преобразовывают данный язык (получивший название "ditroff от device-independent troff — аппаратно-независимый troff) в некоторые данные, которые фактически могут принимать современные графические принтеры — наиболее важным из них (и современным стандартом) является PostScript.

Препроцессоры более интересны, поскольку они фактически расширяют возможности языка troff. Существует 3 распространенных препроцессора:

tbl(1) для создания таблиц,

eqn(1) для текстового представления математических уравнений и

pic(1) для создания диаграмм. Реже используются, но до сих пор сохранились

gm(1) для графики,

refer(1) и

bib(1) для форматирования библиографий. Эквиваленты данных программ с открытыми

исходными кодами поставляются с пакетом

groff. Препроцессор

grap(1) предоставлял довольно гибкое средство для построения графиков; отдельно от

groff существует его реализация с открытым исходным кодом.

Некоторые другие препроцессоры не имеют реализации с открытым исходным кодом и в настоящее время широко не используются. Наиболее известным из них была программа

ideal(1) для форматирования графики. Более новый член данного семейства,

chem(1) отображает формулы химических структур; программа доступна в репозитории netlib Bell Labs[83].

Каждый из описанных препроцессоров представляет собой небольшую программу, которая принимает мини-язык и компилирует его в troff-запросы. Каждый препроцессор распознает разметку, которую необходимо интерпретировать, путем поиска уникального начального и конечного запроса, а любую разметку за пределами таких запросов оставляет неизменной (программа

tbl ищет строки TS/.TE,

pic —

. PS/.PE и т.п.). Таким образом, большинство препроцессоров обычно могут работать в любом порядке, не влияя на другую разметку. Существует несколько исключений: в частности, программы

chem и

grap используют команды программы

pic , и поэтому в конвейерах она должна следовать после них.

cat thesis.ms | chem | tbl | refer | grap | pic | eqn \

 | groff -Tps >thesis.ps

Выше приведен подробный пример конвейера DWB-обработки для гипотетических тезисов, включающих в себя химические формулы, математические уравнения, таблицы, библиографию, графики и диаграммы. (Команда

cat(1) просто копирует свой ввод или содержимое указанного файла на свой вывод; здесь она используется для подчеркивания порядка операций.) На практике современные реализации troff часто поддерживают параметры командной строки, которые способны вызвать, по крайней мере, такие программы, как

tbl(1), eqn(1) и

pic(1) , а поэтому писать такие сложные конвейеры не обязательно. Но даже если бы это понадобилось, такие инструкции для сборки обычно создаются один раз и сохраняются в make-файле или в shell-сценарии для повторного использования.

Разметка документов средствами Documenter's Workbench несколько устарела, однако диапазон проблем, которые решаются препроцессорами, является некоторым показателем мощности модели мини-языков — было бы чрезвычайно трудно встроить эквивалентные

знания в текстовые процессоры класса WYSIWYG. Существует несколько областей, где современные инструментальные связки и способы разметки документов на основе XML, в 2003 году лишь приближаются к возможностям, которыми инструментарий DWB обладал в 1979 году. Эти вопросы подробнее освещаются в главе 18.

Конструктивные идеи, которые дали инструментарию DWB такую мощь, в настоящее время должны быть очевидны. Все инструменты совместно используют общее представление документов в виде текстовых потоков, а система форматирования разбита на независимые компоненты, которые можно отлаживать и совершенствовать по отдельности. Структура конвейеров поддерживает интеграцию с новыми, экспериментальными препроцессорами и постпроцессорами без нарушения работы старых. DWB — модульная и расширяемая конструкция.

Структура инструментария Documenter's Workbench в целом преподносит некоторые уроки того, как связывать несколько специальных языков во взаимодействующую систему. Один препроцессор может быть надстройкой доя другого. Действительно, инструментальные средства DWB были ранними примерами, демонстрирующими мощность каналов, фильтров и мини-языков, которые в дальнейшем во многом повлияли на конструкцию Unix. Конструкции отдельных препроцессоров способны предоставить еще больше примеров того, как выглядит конструкция эффективного мини-языка.

Один из этих уроков отрицательный. Иногда пользователи, пишущие описание в мини-языке, допускают некорректные действия с низкоуровневой troff-разметкой, вставленной вручную. Это может повлечь за собой последствия и ошибки, которые трудно диагностировать, поскольку данные, сгенерированные troff и выходящие из конвейера, не видны, а если бы были видны, то были бы нечитаемыми. Такие ошибки аналогичны ошибкам, которые возникают в коде, когда С-код смешан с фрагментами ассемблера. Было бы лучше, если бы уровни языков были разделены более основательно, если бы это было возможно. Разработчикам мини-языков следует учесть эти проблемы.

Все языки препроцессоров (кроме самой troff-разметки) имеют сравнительно четкий, shell-подобный синтаксис, которые соответствует многим описанным в главе 5 соглашениям о конструкции форматов файлов данных. Существует несколько затруднительных исключений. Особенно выделяется среди них программа

tbl(1) , по умолчанию использующая символ табуляции как разделитель полей между столбцами таблицы, дублирующая неприятные недоработки в конструкции make(1) и вызывающая досадные ошибки, когда редакторы или другие средства невидимо изменяют состав разделителей.

Хотя troff сам по себе представляет собой специализированный императивный мини-язык, одной из идей, которая "проходит" как минимум через 3 мини-языка в DWB, является декларативная семантика: компоновка документа на основе ограничивающих условий. Данная идея также характерна для современных GUI-инструментариев. Вместо того чтобы указывать координаты пикселей для графических объектов, единственное, что действительно требуется сделать — это объявить пространственные взаимозависимости между ними ("элемент управления А расположен выше элемента В, который находится слева от элемента С"), а затем заставить программное обеспечение вычислить наилучшее расположение элементов А, В и С, соответствующее заданным ограничивающим условиям.

В программе

pic(1) данный подход используется для компоновки элементов диаграмм. Диаграмма классификации языков на рис. 8.1 была создана на основе приведенного в примере 8.4[84] исходного

pic -кода, обработанного с помощью команды pic2graph, которая рассматривалась в одном из учебных примеров главы 7.

Это весьма типичная для Unix конструкция мини-языка, и как таковая она имеет несколько интересных моментов даже на уровне синтаксиса. Следует отметить ее сходство с shell-программой: комментарии начинаются с символа #, а синтаксис, очевидно, организован на основе лексем и имеет простейшее возможное соглашение для строк. Разработчик

pic(1) знал, что Unix-программисты ожидают подобный этому синтаксис мини-языков, если не существует значительной и специфической причины не делать этого. В данном случае в полной мере выполняется правило наименьшей неожиданности.Пример 8.4. pic-код для схемы классификации языков

```
#Minilanguage taxonomy (классификация мини-языков)

#

# Base ellipses (основные эллипсы)

define smallellipse {ellipse width 3.0 height 1.5}

M: ellipse width 3.0 height 1.8 fill 0.2

line from M.n to M.s dashed

D: smallellipse() with .e at M.w + (0.8, 0)

line from D.n to D.s dashed

I: smallellipse() with .w at M.e - (0.8, 0)

#

# Captions (подписи)

"" "Data formats" at D.s

"" "Minilanguages" at M.s

"" "interpreters" at I.s

#

# Heads (заголовки)

arrow from D.w + (0.4, 0.8) to D.e + (-0.4, 0.8)

"flat to structured" "" at last arrow.c

arrow from M.w + (0.4, 1.0) to M.e + (-0.4, 1.0)

"declarative to imperative" "" at last arrow.c

arrow from I.w + (0.4, 0.8) to I.e + (-0.4, 0.8)

"less to more general" "" at last arrow.c

#
```

```
# The arrow of loopiness (стрелка развития циклов)

arrow from D.w + (0, 1.2) to I.e + (0, 1.2)

"increasing loopiness" "" at last arrow.c

#

# Flat data files (плоские файлы данных)

"/etc/passwd" ".newsrc" at 0.5 between D.c and D.w

# Structured data files (структурированные файлы данных)

"SNG" at 0.5 between D.c and M.w

# Datafile/minilanguage borderline cases (пограничные случаи файлы данных/мини-язык)

"regexps" "Glade" at 0.5 between M.w and D.e

# Declarative minilanguages (декларативные мини-языки)

"m4" "Yacc" "Lex" "make" "XSLT" "pic" "tbl" "eqn" \

 at 0.5 between M.c and D.e

# Imperative minilanguages (императивные мини-языки)

"fetchmail" "awk" "troff" "Postscript" at 0.5 between M.c and I.w

# Minilanguage/interpreter borderline cases (пограничные случаи

 мини-язык/интерпретатор)

"dc" "bc" at 0.5 between I.w and M.e

# Interpreters (интерпретаторы)

"Emacs Lisp" "JavaScript" at 0.25 between M.e and I.e

"sh" "tcl" at 0.55 between M.e and I.e

"Perl" "Python" "Java" at 0.8 between M.e and I.e
```

Вероятно, больших усилий не потребуется, чтобы понять, что первая строка кода представляет собой определение макрокоманды. В последующих ссылках на smallellipse() инкапсулирован повторяющийся элемент диаграммы. Назначение команды arrow также очевидно.

Используя все это как подсказку и глядя на реальную диаграмму, несложно выяснить значение остальных элементов синтаксиса (позиционных ориентиров, таких как M.s, и конструкций, подобных last arrow или at 0.25 between M.e and I.e, или добавление смещения вектора). Как и Glade-разметка, а также

m4- код, пример, подобный данному, может прояснить многое в языке без каких-либо ссылок на руководства (к сожалению, свойство компактности для

troff(1) -разметки

не характерно).

Пример программы

pic(1) отражает общую для мини-языков идею конструкции, которая также отражается в Glade — использование интерпретатора мини-языка для инкапсуляции некоторой формы логических расчетов на основе ограничивающих условий и превращения ее в действия. Программу

pic(1) , в сущности, можно было бы рассматривать скорее как императивный, а не декларативный язык; в ней имеются элементы обоих видов, и дискуссия быстро переросла бы теологическую.

Комбинация макросов с компоновкой на основе ограничивающих условий позволяет программе

pic(1) выражать структуру диаграмм таким способом, который недоступен для более современных векторных разметок, таких как SVG. Следовательно, благоприятно, то, что одним из следствий конструкции Documenter's Workbench является то, что она относительно упрощает использование программы

pic(1) за пределами среды DWB. Сценарий pic2graph, использованный в качестве учебного примера в главе 7, был специально создан для достижения этой цели с помощью модернизированных PostScript-возможностей

groff(1) как промежуточный этап на пути к современному растровому формату.

Более четким решением является утилита

pic2plot(1) , распространяемая с пакетом GNU plotutils, в которой использована внутренняя модульность кода GNU

pic(1). Код был разделен на клиентскую часть, выполняющую синтаксический анализ, и серверную часть, генерирующую troff-разметку. Обе части взаимодействовали посредством уровня чертежных примитивов. Поскольку данная конструкция подчинялась правилу модульности, программисты

pic2plot(1) имели возможность отделить этап синтаксического анализа GNU

pic и реконструировать чертежные примитивы с помощью современной библиотеки для построения графиков. Однако их решение имеет один недостаток. Текст на выходе генерируется со встроенными в

pic2plot шрифтами, которые не соответствуют шрифтам troff.

8.2.7. Учебный пример: синтаксис конфигурационного файла

fetchmail

Рассмотрим пример 8.5.

Конфигурационный файл может рассматриваться как императивный мини-язык. Существует предполагаемый поток выполнения: повторяющаяся, циклическая обработка списка команд опроса ("засыпающая" на время в конце каждого цикла) и последовательный сбор почты с

каждого из указанных узлов для каждого пользователя, связанного с определенными узлами. Данный язык далек от универсальных языков. Все, что он способен делать, — создавать последовательность команд опроса серверов.

Как и в случае с программой

pic(1), данный мини-язык можно рассматривать как объявления либо как слабый императивный язык и бесконечно спорить об отличиях. С одной стороны, в нем нет ни условных операторов, ни рекурсии, ни циклов. Фактически он вообще не имеет явных управляющих структур. С другой стороны, он описывает скорее действия, чем зависимости, что отличает его от исключительно декларативного синтаксиса, подобного GUI-описаниям Glade.Пример 8.5. Синтетический код fetchmailrc

#Опрашивать данный узел первым в цикле.

poll pop.provider.net proto pop3

 user "jsmith" with pass "secret1" is "smith" here

 user jones with pass "secret2" is "jjones" here with options keep

Опрашивать данный узел вторым

poll billywig.hogwarts.com with proto imap:

 user harry_potter with pass "floo" is harry_potter here

Опрашивать данный узел третьим в цикле.

Пароль будет взят из файла ~/.netrc

poll mailhost.net with proto imap:

 user esr is esr here

Конфигурационные мини-языки для сложных программ часто переходят эту границу. Данный факт подчеркивается здесь потому, что отсутствие явных управляющих структур в императивном мини-языке может быть колоссальным упрощением, если это позволяет предметная область.

Примечательной особенностью синтаксиса .fetchmailrc является использование необязательных ключевых слов, которые поддерживаются просто для того, чтобы язык спецификаций более походил на английский язык. Ключевые слова "with" и однократное употребление слова "options" в примере фактически не являются обязательными, но позволяют упростить описания для чтения.

Традиционно подобный синтаксис называется

синтаксическим сахаром (syntactic sugar) . Данному термину сопутствует известное высказывание о том, что "синтаксический сахар вызывает рак двоеточий"[85]. Действительно, чтобы синтаксический сахар не создавал трудностей больше, чем может решить проблем, его необходимо использовать умеренно.

В главе 9 показано, как создание программ, управляемых данными, способствует изящному решению проблемы редактирования конфигурационных файлов

fetchmail с помощью графического интерфейса.

8.2.8. Учебный пример:

awk

Мини-язык

awk является инструментальным средством Unix старой школы, прежде широко используемым в shell-сценариях. Как и

m4 , утилита

awk предназначена для написания небольших, но выразительных программ для преобразования текстового ввода в текстовый вывод. Версии утилиты поставляются со всеми Unix-системами. Некоторые из них реализованы с открытым исходным кодом. Команда info gawk в командной строке Unix весьма вероятно позволит получить справочную документацию по программе.

Программы, написанные на

awk , состоят из пар шаблон/действие. Каждый шаблон представляет собой

регулярное выражение ; эта концепция подробно описывается в главе 9. После запуска awk-программа последовательно анализирует все строки во входном файле. Каждая строка по порядку сравнивается с парой шаблон/действие. Если шаблон соответствует строке, то осуществляется связанное с шаблоном действие.

Каждое действие кодируется на языке, подобном подмножеству языка C, с переменными, условными операторами, циклами и онтологией типов, включая целые числа, строки и (в отличие от C) словари[86].

Язык действий

awk является языком Тьюринга и позволяет считывать и записывать файлы. В некоторых версиях он также позволяет открывать и использовать сетевые сокеты. Однако

awk главным образом используется как генератор отчетов, особенно для интерпретации и предварительной обработки табличных данных. Он редко используется автономно, но часто встраивается в сценарии. В главе 9, в учебном примере по созданию HTML-документа имеется пример awk-программы.

Учебный пример

awk приведен в этой книге, чтобы подчеркнуть, что данный язык

не является моделью для подражания. Фактически с 1990 года

awk почти совершенно вышел из употребления. На смену ему пришли языки сценариев новой школы, особенно Perl, который явно предназначался для того, чтобы полностью вытеснить

awk. Причины достойны внимания, поскольку они поучительны для разработчиков мини-языков.

Язык

awk первоначально разрабатывался как небольшой, выразительный язык специального назначения для создания отчетов. К сожалению, его соотношение сложность-мощность оказалось неудачным. Язык действий некомпактен, а шаблонно-управляемая структура, внутри которой он содержится, не позволяет применять его широко. Данный язык унаследовал худшие черты обоих миров. Кроме того, языки сценариев новой школы могут решать все задачи, решаемые

awk. Эквивалентные программы, написанные на этих языках, обычно также, если не лучше, читабельны.

Язык

awk вышел из употребления также вследствие того, что более современные оболочки обладают средствами вычислений с плавающей точкой, ассоциативными массивами, поддержкой регулярных выражений и средствами обработки подстрок, поэтому эквивалентные небольшим

awk- сценариям программы могут быть реализованы без издержек создания процесса.Дэвид Корн.

В течение нескольких лет после выхода языка Perl в 1987 году,

awk оставался конкурентоспособным просто потому, что имел меньшую и более быструю реализацию. Однако по мере того как стоимость вычислительных циклов и памяти падала, экономические причины для привлекательности языка специального назначения, который сравнительно экономно использовал оба ресурса, теряли свою силу. Программисты для реализации awk-подобных функций все более отдавали предпочтение Perl или (позднее) языку Python, вместо того, чтобы удерживать в памяти два различных языка сценария[87]. К 2000 году

awk стал для большинства Unix-хакеров старой школы немногим больше, чем воспоминание, но не самое дорогое.

Снижение цен изменило компромиссы проектирования мини-языков. Ограничение возможностей конструкции ради компактности, возможно, до сих пор является хорошей идеей, но такое же ограничения в целях экономии аппаратных ресурсов — идея неудачная. Со временем аппаратные ресурсы становятся дешевле, а пространство в памяти программистов дороже. Современные мини-языки могут быть универсальными и некомпактными, или специализированными и очень компактными, но специализированные и некомпактные просто не выдержат конкуренции.

8.2.9. Учебный пример: PostScript

PostScript — мини-язык, специализацией которого является описание форматированного текста и графики для графических устройств. Данный язык был импортирован в Unix. Он основывался на разработке легендарного центра "Xerox Palo Alto Research Center", созданной во время появления первых лазерных принтеров. В течение нескольких лет после выхода первой коммерческой версии в 1984 году, PostScript оставался доступным только как частный продукт Adobe, Inc. и главным образом ассоциировался с компьютерами Apple. PostScript был клонирован на условиях лицензионного соглашения, очень близкого к лицензиям на открытые исходные коды, и с тех пор стал стандартом де-факто для управления принтерами в операционной системе Unix. Версия с полностью открытым исходным кодом поставляется с

большинством современных Unix-систем[88]. Также доступно подробное техническое введение в PostScript[89].

PostScript обладает некоторым функциональным сходством с разметкой troff. Оба языка предназначены для управления принтерами и другими графическими устройствами, и оба обычно генерируются программами или пакетами макрокоманд, а не вручную. Однако тогда как запросы troff являются быстро созданным набором кодов для управления форматом, PostScript был спроектирован снизу вверх как язык и является гораздо более выразительным и мощным. Главное из того, что делает PostScript, — это алгоритмические описания изображений, имеющие гораздо меньшие размеры, чем представленные ими растровые изображения, и поэтому требующие меньше пространства для хранения и меньшей полосы пропускания при передаче.

PostScript является явным языком Тьюринга, поддерживающим условные операции, циклы, рекурсию и именованные процедуры. Онтология типов включает в себя целые и действительные числа, строки и массивы (каждый элемент массива может иметь любой тип), но не имеет эквивалента структур. Технически PostScript является языком, работающим со стеками. Аргументы примитивных процедур (операторов) PostScript обычно извлекаются из магазинного стека аргументов, а результат (или результаты) возвращаются обратно в стек.

Существует около 40 базовых операторов (при том, что общее их приблизительное количество — 400). Большую часть работы выполняет оператор show, который отображает строку на странице. Другие операторы устанавливают текущий шрифт, изменяют цвет, рисуют линии, дуги или кривые Безье, окрашивают закрытые области, устанавливают области отсечения, а также выполняют другие операции. Подразумевается, что интерпретатор PostScript транслирует данные команды в растровые изображения для передачи на экран или печатный носитель.

Остальные PostScript-операторы реализуют арифметические операции, управляющие структуры и процедуры. Они позволяют выражать повторяющиеся или стереотипные изображения (такие как текст, составленный из повторяющихся графических форм знаков) в виде программ, объединяющих изображения. Часть эффективности PostScript связана с тем фактом, что PostScript-программы для печати текста или простой векторной графики являются гораздо менее громоздкими, чем растровые изображения, в которые преобразовывается текст или векторы. Кроме того, PostScript-программы независимы от разрешающей способности устройств и быстрее передаются по сетевому кабелю или последовательной линии.

Исторически PostScript-интерпретация на основе стеков имеет сходство с языком FORTH, который первоначально предназначался для управления приводами телескопов в реальном времени и имел кратковременную популярность в 80-х годах прошлого века. Языки обработки стеков отличаются превосходной поддержкой чрезвычайно плотного, экономичного кода и печально известны тем, что их трудно читать. Для PostScript характерны обе эти особенности.

PostScript часто реализуется в виде встроенного в принтер программного обеспечения. Ghostscript, реализация PostScript с открытым исходным кодом транслирует PostScript в различные графические форматы и (более слабые) языки управления принтерами. Большая часть остального программного обеспечения обрабатывает PostScript как окончательный формат вывода, который предназначен для передачи PostScript-совместимому графическому устройству, а не для редактирования или просмотра.

PostScript (в оригинальном или упрощенном варианте EPSF с объявленным вокруг него обрамлением, позволяющим встраивать его в другие графические форматы) является очень хорошо спроектированным примером специализированного языка управления и, как модель,

заслуживает внимательного изучения. Он является компонентом других стандартов, например, PDF (Portable Document Format — формат для переносимых документов).

8.2.10. Учебный пример: утилиты

bc и

dc

Впервые утилиты

bc(1) и

dc(1) рассматривались в главе 7 как учебный пример вызова с созданием подоболочки. Они также являются примерами узкоспециальных мини-языков императивного типа.

dc — старейший язык в Unix. Он был создан на компьютере PDP-7 и перенесен на PDP-11 еще до того, как была перенесена (сама) UnixКен Томпсон.

Предметной областью данных языков являются арифметические вычисления неограниченной точности. Другие программы могут использовать их для выполнения таких вычислений, "не заботясь" о необходимых для этого специальных методиках.

Фактически первоначальная мотивация для создания dc не была связана с разработкой универсального интерактивного калькулятора, для которого было бы достаточно простой программы, поддерживающей вычисления с плавающей точкой. Мотивирующим фактором была давняя заинтересованность Bell Labs в численном анализе:

точному вычислению констант для численных алгоритмов весьма способствует возможность работать с более высокой точностью, чем та, которую использует сам алгоритм. Отсюда и арифметика с неограниченной точностью в dc.Генри Спенсер.

Как и в случае SNG- и Glade-разметки одним из преимуществ обоих языков является их простота. Однажды узнав о том, что

dc(1) — калькулятор с обратной польской записью, а

bc(1) — калькулятор с алгебраической записью, пользователь найдет новыми очень немногие сведения об интерактивном использовании любого

из двух языков. Важность правила наименьшей неожиданности в интерфейсах повторно рассматривается в главе 11.

В обоих мини-языках имеются условные операции и циклы; они являются языками Тьюринга, но имеют очень ограниченную онтологию типов, включающую только целые числа неограниченной точности и строки, что ставит их между интерпретируемыми мини-языками и полными языками сценариев. Функции программирования реализованы так, чтобы не нарушать обычное использование программы в качестве калькулятора; в действительности, большинство пользователей

dc/bc , вероятно, не подозревают о существовании этих функций.

Обычно программы

dc/bc используются в диалоговом режиме, но их способность поддерживать библиотеки пользовательских процедур предоставляет им дополнительный вид служебных функций — программируемость. Фактически данное свойство является наиболее важным преимуществом императивных мини-языков, которое, как отмечалось в учебном примере по инструментарию Documenter's Workbench, должно быть очень мощным независимо от того, является ли обычный режим работы программы диалоговым. Данные языки можно использовать для написания высокоуровневых программ, которые включают в себя специализированную логику.

Поскольку интерфейс программ

dc/bc прост (передается строка, содержащая выражение, в ответ на которую возвращается строка, содержащая значение), другие программы и сценарии могут легко получит доступ ко всем этим возможностям, вызывая

dc/bc как подчиненные процессы. Ниже приводится один известный пример (см. пример 8.6), реализация шифра с открытым ключом Ривеста-Шамира-Адельмана (Rivest-Shamir-Adelman) на Perl, которая широко публиковалась в подписях почтовых сообщений и на футболках как протест против введенного в США в 1995 году ограничения на экспорт криптографических средств. Данный сценарий для выполнения необходимых арифметических расчетов с неограниченной точностью вызывает

dc с созданием подоболочки.Пример 8.6. RSA-реализация с помощью утилиты

dc

```
print pack"C*",split/\D+/,`echo "16iII*o\U@{$/=$z;[(pop,pop,unpack "H*",
&lt;&gt;)]}\EsMsKsN0[1N*11K[d2%Sa2/d0&lt;X+d*1MLa^*1N%0]dsXx++\
1M1N/dsM0&lt;J]dsJxp"|dc`
```

8.2.11. Учебный пример: Emacs Lisp

Вместо того чтобы просто запускать интерпретируемый язык специального назначения в качестве подчиненного процесса для решения специфических задач, его можно использовать как основу для всей структуры. Преимущества и недостатки данного подхода рассматриваются в главе 13. Запросы troff были его ранним примером. Одним из самых известных и наиболее мощных современных примеров является редактор

Emacs . Он создан на основе диалекта языка Lisp с примитивами как для описания действий в буферах редактирования, так и управления подчиненными процессами.

Тот факт, что Emacs построен вокруг мощного языка для описания редактирующих действий или клиентской части для других программ, означает, что он, кроме обычного редактирования, может применяться для многих других целей. Применение развитой, специализированной логики Emacs для повседневной разработки программ (компиляции, отладки, контроля версий) рассматривается в главе 15. "Режимы" Emacs представляют собой пользовательские библиотеки, т.е. программы, написанные на Emacs Lisp, которые приспосабливают редактор для решения специфической задачи, обычно (но не обязательно) связанной с редактированием.

Таким образом, существуют специализированные режимы, которые распознают синтаксис большого числа языков программирования и разметки, таких как SGML, XML и HTML. Однако

многие пользователи также используют Emacs-режимы для отправки и получения электронной почты (в таких режимах в качестве подчиненных процессов используются почтовые утилиты Unix) или новостей Usenet. Emacs может служить в качестве Web-браузера или клиентской части для различных программ интерактивного общения. Существует также пакет для составления расписаний, собственная программа-калькулятор для Emacs и даже весьма широкий выбор игр, написанных как режимы Emacs Lisp.

8.2.12 Учебный пример: JavaScript

JavaScript — язык с открытым исходным кодом, спроектированный для внедрения в C-программы. Несмотря на то, что он также встраивается в Web-серверы, первоначальным и наиболее известным его проявлением является клиентский JavaScript, который позволяет встраивать исполняемый код в Web-страницы, просматриваемые любым поддерживающим его браузером. Здесь рассматривается именно эта версия языка.

JavaScript — интерпретируемый язык, полностью попадающий под определения языка Тьюринга, с целыми, действительными числами, булевыми операторами, строками и легковесными, основанными на словарях, объектами, которые имеют сходство с объектами языка Python. Значения типизированы, но переменные могут иметь любой тип. Преобразование типов осуществляется автоматически во многих средах. Синтаксически JavaScript подобен языку Java с некоторым влиянием со стороны Perl и содержит функции для работы с Perl-подобными регулярными выражениями.

Несмотря на все указанные особенности, клиентская версия JavaScript не является полностью универсальным языком. Его возможности жестко ограничены в целях предотвращения атак на браузеры через Web-страницы, содержащие JavaScript-код. Язык принимает входные данные от пользователя и генерирует или модифицирует Web-страницы, но непосредственно не может изменять содержимое дисковых файлов и открывать собственные сетевые соединения.

С течением времени данный язык стал более универсальным и менее связанным со своей клиентской средой. Это закономерное развитие любого удачного специализированного языка. Клиентский вариант JavaScript в настоящее время взаимодействует со своим окружением путем чтения и записи значений в один специальный объект, который называется документной объектной моделью браузера (Document Object Model, DOM). Данный язык до сих пор имеет некоторые устаревшие API-интерфейсы для связи с браузером, которые не работают через DOM, однако они нежелательны, отсутствуют в стандарте для JavaScript ECMA-262 и могут не поддерживаться в будущих версиях.

Стандартным справочником по JavaScript является книга

"JavaScript: The Definitive Guide" [20]. Исходный код доступен в Web[90]. Язык JavaScript представляет собой интересный пример для изучения по двум причинам. Во-первых, он максимально близок к универсальному языку, фактически не являясь таковым. Во-вторых, связь между клиентским JavaScript и средой браузера через единственный DOM-объект хорошо спроектирована и может послужить моделью для других ситуаций встраивания.

8.3. Проектирование мини-языков

В каких ситуациях целесообразна разработка мини-языка? Выше отмечалось, что мини-языки предоставляют способ перемещения спецификаций проблем на более высокий уровень, а в нескольких учебных примерах это наблюдение подтверждалось фактами. Оборотная сторона данного явления — использование мини-языка, вероятно, будет хорошим подходом всякий раз, когда примитивы предметной области просты и стереотипны, а способы их вероятного применения пользователями изменчивы.

Некоторые родственные идеи можно найти в описании моделей проектирования "Alternate Hard And Soft Layers" <http://www.c2.com/cgi/wiki?Alternate-HardAndSoftLayers> и "Scripted Components" <http://www.doc.ic.ac.uk/~np2/patterns/scripting/scripting.html>.

Интересное обозрение стилей и методик разработки с помощью мини-языков приведено в статье

"Notable Design Patterns for Domain-Specific Languages" [77].

8.3.1. Определение соответствующего уровня сложности

Первым важным моментом, о котором необходимо помнить при разработке мини-языка, является, как обычно, сохранение по возможности простой конструкции. Диаграмма классификации, которая использовалась для организации учебных примеров, выражает иерархию сложности. Необходимо удерживать конструкцию как можно ближе к левой границе схемы. Если с проблемой можно справиться путем разработки структурированного файла данных вместо использования мини-языка, который стремится модифицировать внешние данные, в случае если он является интерпретируемым, то следует любыми средствами реализовать эту возможность.

Одной чрезвычайно прагматичной причиной, по которой следует продолжать усердно работать со структурированными данными, а не с мини-языком, является то, что в сетевом мире средства встроенного мини-языка являются предметом неправильного использования, которое может причинять беспокойство или даже быть опасным. JavaScript — главный экземпляр в категории "причиняющих беспокойство". Разработчики данного языка не предвидели того, что он будет использоваться для всплывающей рекламы настолько навязчиво, что создаст потребность в функциях браузера, подавляющих интерпретацию JavaScript.

Макровирусы Microsoft Word показывают, как мини-язык может стать действительно опасной брешью в безопасности системы, простои и потери производительности от которой ежегодно обходятся в миллиарды долларов. Полезно отметить, что несмотря на существование в мире как минимум 20 млн. пользователей[91] Unix, в Unix-среде никогда не было вспышек макровирусов, характерных для Windows. Существует множество причин этому, включая принципиально лучшую с точки зрения безопасности конструкцию Unix; но по крайней мере одной причиной является тот факт, что почтовые агенты в Unix по умолчанию

не обрабатывают исполняемое содержимое в документах, просматриваемых пользователем [92].

Если существует какая-либо возможность того, что пользователи приложения могут запускать программы из неблагонадежных источников, рискованные функции мини-языка приложения могут в конце концов привести к необходимости его подавления. Языки, подобные Java и JavaScript, явно изолируются в

"песочнице", т.е. они имеют ограниченный доступ к своему окружению. Это сделано не только для того, чтобы упростить их конструкцию, но и для того, чтобы воспрепятствовать потенциально деструктивным операциям со стороны ошибочного или злонамеренного кода.

С другой стороны, большое количество неудачных конструкций были неумело созданы разработчиками, которые оказались не способны принять тот факт, что они нуждаются скорее в мини-языке, чем в формате файлов данных. Слишком много случаев, когда языковые функции были вставлены с опозданием. Двумя наиболее общими симптомами данной проблемы являются слабые, узкоспециализированные управляющие структуры и неразвитые средства для объявления процедур или полное отсутствие таких средств.

Рискованно разрабатывать мини-языки, которые только отчасти являются языками Тьюринга. Если разработчик решается на это, то высока вероятность того, что когда-нибудь в будущем какой-нибудь сообразительный программист придет к необходимости заставить данный язык выполнять циклы и условные операции. Поскольку это можно будет сделать только "туманным путем", он создаст неясный код, который в краткосрочной перспективе, возможно, будет легко обслуживаемым, но, вероятно, будет кошмаром для тех, кто впоследствии будет с ним работать.

Эстетичность и мощь мини-языков заслуживают высокой оценки, но вместе с тем в них скрывается немало ловушек. Существуют конструкции, в которых целесообразно использовать восходящее связывание множества низкоуровневых служб и заботиться об их организации после изучения предметной области. Одним из достоинств мини-языков является то, что они могут способствовать созданию хорошей конструкции без восходящего программирования, позволяя разработчику передвинуть некоторые нисходящие решения в управляющую логику программ, написанных на данном мини-языке. Однако если применить восходящий подход к конструированию

самого мини-языка, то в результате, вероятно, получится уродливый синтаксис, отражающий слабость языка и плохо продуманную реализацию.

Существует множество моментов в конструировании мини-языков, где мелкие альтернативные решения создают значительные различия в эксплуатационных качествах и легкости использования инструмента.

Для разработчика языка хорошим принципом является рассмотрение альтернативы сообщениям об ошибке. Если в намерениях программиста существует действительная неоднозначность, то сообщение об ошибке целесообразно, однако во многих случаях намерения очевидны, и будет великим благом заставить язык просто выполнять правильные действия. Хорошим примером является C, принимающий дополнительную запятую в конце списка инициализатора массива, что значительно упрощает как редактирование, так и машинную генерацию инициализаторов массива. Контрпримером является придирчивость различных HTML-анализаторов, особенно их обыкновение бесшумно отбрасывать части документа из-за тривиальной ошибки верстки. Стив Джонсон.

В данном вопросе, как и в других, хороший вкус и инженерное мышление невозможно заменить ничем. Разрабатывая мини-язык, не следует делать это наполовину. Декларативные мини-языки должны иметь очевидный, последовательный языковой синтаксис, облегчающий их чтение. В императивных языках необходимо добавить полный диапазон управляющих структур, который адаптирован из языковых моделей, с которыми, пользователи разрабатываемого мини-языка, вероятно, знакомы. О языке необходимо думать

как о языке, спрашивая себя: "будет ли удобно программировать на нем?" и даже "приятно ли будет смотреть на данную конструкцию?". Здесь, как и в других областях разработки

программного обеспечения, применим принцип Дэвида Гелентера: красота — основная защита против сложности.

8.3.2. Расширение и встраивание языков

Один из фундаментально важных вопросов заключается в том, возможно ли реализовать мини-язык путем расширения или встраивания существующего языка сценариев. Нередко такой подход является правильным путем к императивному мини-языку, но гораздо менее верным к декларативному.

Иногда можно написать императивный язык путем простого кодирования служебных функций в интерпретируемый язык, который далее в целях освещения данной темы называется "базовым" (host) языком. Программы, написанные на таком мини-языке, являются просто сценариями, которые загружают служебную библиотеку и используют управляющие структуры и другие средства базового языка как каркас. Каждое средство, имеющееся в базовом языке, избавляет от необходимости писать собственное аналогичное средство.

Данный способ является простейшим для написания мини-языка. Lisp-программисты старой школы (включая самого автора) предпочитают данный прием и интенсивно его используют. Он лежит в основе конструкции редактора

Emacs и открывается заново в языках сценариев новой школы, таких как Tcl, Python и Perl. Однако такой подход имеет свои недостатки.

Базовый язык может оказаться неспособным предоставить интерфейс к необходимой библиотеке кода. Или его онтология типов данных может оказаться неадекватной для необходимого вида вычислений. Или измерения покажут слишком низкий уровень производительности прототипа. В таких ситуациях обычным решением является программирование на C (или C++) и интеграция результатов в создаваемый мини-язык.

Вариант расширения языка сценариев с помощью C-кода или внедрения языка сценариев в C-программу зависит от существования предназначенного для этих целей языка сценариев. Расширить язык сценариев можно, заставив его динамически загружать C-библиотеку или модуль так, чтобы точки входа C стали видимыми, как функции в расширенном языке. Для того чтобы встроить язык сценария в C-программу, необходимо отправлять команды экземпляру интерпретатора и получать результаты, как значения в C.

Обе методики также зависят от способности перемещать данные между онтологией типов C и онтологией типов используемого языка сценариев. Некоторые языки сценариев, в целях поддержки такой возможности разработаны снизу вверх. Одним из них является Tcl, который рассматривается в главе 14, другим —

Guile, диалект с открытым исходным кодом Lisp-варианта Scheme.

Guile поставляется в виде библиотеки и специально предназначен для внедрения в C-программы.

Вполне возможно (хотя до сих пор довольно трудно) расширять или встраивать язык Perl. Очень просто расширять Python и немного сложнее внедрять его; C-расширения в мире Python используются особенно интенсивно. Язык Java имеет интерфейс для вызова "собственных методов" в C, хотя практические результаты этого бесперспективны, поскольку нарушается переносимость. Большинство версий командного интерпретатора Unix (shell) не

предназначены для внедрения или расширения, однако оболочка Korn (ksh93 и боле поздние версии) является заметным исключением.

Существует множество причин не надстраивать создаваемый императивный мини-язык над существующим языком сценариев. Одна из весомых причин заключается в необходимости реализовать собственный, специальный грамматический аппарат для проверки ошибок. В таком случае следует рассмотреть приведенные ниже рекомендации по применению утилит

yacc и

lex .

8.3.3. Написание специальной грамматики

Для декларативных мини-языков основной вопрос состоит в том, следует ли использовать XML в качестве основного синтаксиса и определять грамматику как тип XML-документа. Вполне возможно, что такой подход верен для сложно структурированных декларативных мини-языков, однако здесь также актуальны предостережения, изложенные в главе 5, о конструкции форматов файлов данных — XML может быть излишним. Если XML не используется, следует соблюдать правило наименьшей неожиданности, поддерживая описанные Unix-соглашения для файлов данных (простой синтаксис на основе лексем, поддержка C-соглашений об использовании обратной косой черты и т.д.).

Если специальная грамматика действительно требуется, то утилиты

yacc и

lex (или их локальный эквивалент в используемом языке), вероятно, будут наилучшими помощниками, кроме тех случаев, когда грамматика используемого языка настолько проста, что ручное кодирование рекурсивного нисходящего синтаксического анализатора представляет собой тривиальную задачу. Даже тогда утилита

yacc может предоставить более надежное устранение ошибок, а модифицировать yacc-спецификацию по мере развития синтаксиса языка будет проще. В главе 9 рассматриваются производные от

yacc и

lex инструменты, доступные в языках различной реализации.

Даже в случае принятия решения о реализации собственного синтаксиса рекомендуется рассмотреть возможную выгоду от повторного использования имеющихся инструментальных средств. Если требуются макросредства, следует учесть, что предобработка средствами

m4(1) может быть правильным решением. Однако, прежде всего, необходимо учесть предостережения, приведенные в следующем разделе.

8.3.4. Проблемы макросов

Средства макрорасширения были излюбленной тактикой разработчиков языков в ранней

Unix. Язык C, несомненно, имеет такое средство. Кроме того, они обнаруживаются в некоторых более сложных мини-языках специального назначения, таких как

pic(1). Препроцессор

m4 предоставляет общее средство для реализации макрорасширяющих препроцессоров.

Макрорасширение просто определить и реализовать, а также осуществить с его помощью множество изящных и нетривиальных технических приемов. На ранних разработчиков, по-видимому, оказывал влияние опыт ассемблера, в котором макросредства часто были единственным механизмом, доступным для структурирующих программ.

Преимуществом макрорасширения является то, что оно не имеет сведений о синтаксисе, лежащем в основе базового языка, и может применяться для расширения данного синтаксиса. К сожалению, данным преимуществом очень легко злоупотребляют, создавая непрозрачный, непредсказуемый код, который является богатым источником тяжело определяемых ошибок.

Для языка C классическим примером такой проблемы является макрос, подобный следующему.

#define max(x, y) x > y ? x : y

Данный макрос создает как минимум две проблемы. Одна из них заключается в том, что он может вызвать непредсказуемые результаты, в случае если один из аргументов является выражением, включающим в себя оператор меньшего приоритета, чем > или ?:. Рассмотрим выражение max(a = b, ++c). Если программист забыл, что max является макросом, то он будет ожидать, что присваивание a = b и преинкрементная операция c с будут выполнены до того, как результирующие значения будут переданы max в качестве аргументов.

Однако это не так. Вместо этого препроцессор преобразует данное выражение в a = b > ++c ? a = b : ++c, которое правила приоритета компилятора C заставляют интерпретировать как a = (b > ++c ? a = b : ++c). Результат будет присвоен a.

Подобное неверное взаимодействие можно предотвратить, кодируя определение макроса более безопасно.

#define max(x, y) ((x) > (y) ? (x) : (y))

С таким определением выражение будет развернуто как ((a = b) > (++c) ? (a = b) : (++c)), что решает одну проблему, однако следует заметить, что переменная c может быть инкрементирована дважды. Существуют менее очевидные варианты данной проблемы, такие как передача макросу вызова функции с побочными эффектами.

Как правило, взаимодействие между макросами и выражениями с побочными эффектами может привести к неудачным результатам, которые трудно диагностировать. Макропроцессор C умышленно создан легковесным и простым. Более мощные макропроцессоры способны действительно вызвать более серьезные проблемы.

Язык форматирования TEX (см. главу 18) хорошо иллюстрирует общую проблему.

TeX — умышленно разрабатывался как язык Тьюринга (в нем имеются условные операции, циклы и рекурсия), однако, несмотря на то, что его можно заставить делать поразительные вещи, TEX-код часто нечитабельный и трудный в отладке. Исходные коды для LATEX, наиболее широко используемого TEX-макропакета, являются поучительным примером: они созданы в очень хорошем TEX-стиле, но даже несмотря на это их крайне трудно понять.

Менее значительная по сравнению с описанной проблема заключается в том, что макрорасширение склонно усложнять диагностику ошибок. Процессор базового языка создает отчеты об ошибках, относящиеся к развернутому макросом тексту, а не к оригинальному коду, который просматривает программист. Если связь между ними затемняется макрорасширением, то, возможно, что созданные диагностические отчеты будет очень трудно связать с фактическим местом возникновения ошибки.

Данная проблема особенно характерна для препроцессоров и макросов, которые могут иметь многострочные расширения, условно включать или исключать текст, или иным путем изменять количество строк в развернутом тексте.

Каскады макрорасширений, встроенные в язык, могут выполнять самокоррекцию, обрабатывая номера строк так, чтобы создать ссылки на текст до расширения. Так работает, например, макросредство утилиты

pic(1) . Данную проблему гораздо труднее решить, когда макрорасширение выполняется препроцессором.

В препроцессоре C проблема решается путем создания директив #line каждый раз, когда выполняется включение или многострочное расширение. Ожидается, что C-компилятор интерпретирует их и соответственно скорректирует номера строк в отчетах об ошибках. К сожалению, в макропроцессоре

m4 подобное средство отсутствует.

Выше были описаны причины для крайне осторожного использования макрорасширений. Один из долгосрочных уроков опыта Unix заключается в том, что макросы склонны создавать больше проблем, чем решают. Современные конструкции языков и мини-языков отходят от их использования.

8.3.5. Язык или протокол прикладного уровня

Не менее важно определить, будут ли другие программы интерактивно вызывать ядро мини-языка в качестве подчиненного процесса. Если это так, то конструкция, вероятно, должна меньше походить на диалоговый язык для взаимодействия с человеком и быть более похожей на протоколы прикладного уровня, рассмотренные в главе 5.

Основное отличие заключается в том, насколько тщательным является обозначение границ транзакций. Люди хорошо определяют, где заканчивается диалоговый вывод CLI-интерфейса и где находится приглашения для следующего ввода. Человек может использовать контекст, для того чтобы определить, что значительно, а что следует проигнорировать. Компьютерным программам гораздо сложнее справиться с этой задачей. Без однозначных маркеров окончания вывода или указания его точной длины они не способны определить, когда необходимо прекратить чтение.

Еще хуже, когда ввод программы буферизируется (часто непреднамеренно, как в stdio). Программа, которая очевидно прекращает чтение в верной позиции, может, несмотря на это, впоследствии принять ввод. Дуг Макилрой.

Программы, в которых главные процессы пытаются интерактивно обмениваться данными с подчиненными мини-языками, где данная проблема тщательно не проработана, подвержены взаимоблокировкам при рассинхронизации главного и подчиненного процессов (проблема,

впервые упомянутая в главе 7).

Существуют обходные пути для управления мини-языками, разработанными не так тщательно. Прототипом для большинства из них является Tcl-пакет

expect . Данный пакет облегчает диалог с CLI-интерфейсами. Он создан на основе следующей операции: чтение данных подчиненного процесса до тех пор, пока либо не будет найдено соответствие заданному регулярному выражению, либо пока не истечет определенное время. Используя это (и, конечно, операцию отправки данных подчиненному процессу), нередко можно сконструировать главные программы, осуществляющие надежные диалоги с подчиненными процессами, даже когда последние к этому не приспособлены.

В других языках доступны эквиваленты пакета

expect . Полезную информацию можно найти в Web, используя в качестве поисковой фразы название языка с дополнительными ключевыми словами "Tcl expect". Однако для разработчика мини-языка было бы недальновидно предполагать, что все пользователи являются специалистами по использованию пакета

expect . Даже если они являются таковыми, данный подход является дополнительным связующим уровнем и местом возникновения ошибок.

При разработке мини-языка следует помнить о данной проблеме. Хорошей идеей может быть добавление опции, которая изменяет диалоговое поведение, и ответы становятся более похожими на протокол прикладного уровня с недвусмысленными разделителями конца вывода и аналогом заполнения байтами.

9

Генерация кода: повышение уровня спецификации

Попавший в тупик программист… часто может сделать больше, отходя от кода, останавливаясь и анализируя имеющиеся данные. Представление является сущностью программирования.

"The Mythical Man-Month", юбилейное издание (1975—1995) —Фред Брукс

В главе 1 отмечалось, что людям гораздо удобнее мысленно представить себе данные, чем анализировать управляющую логику программы. Для того чтобы понять это, следует сравнить выразительность и дидактическую силу диаграммы 50-узлового дерева указателей с блок-схемой программы, состоящей из 50 строк. Или (что еще лучше) диаграмму инициализатора массива, выражающего таблицу преобразования, с эквивалентным оператором выбора. Различия в прозрачности и ясности поразительны[93].

Данные более понятны, чем программная логика. Это верно независимо от того, являются ли данные обычной таблицей, декларативным языком разметки, системой шаблонов или набором макросов, которые расширяют программную логику. Хорошей практикой является перемещение как можно больше сложности конструкции из процедурного кода в данные, а также выбор форм представления данных, которые удобны для человека, осуществляющего сопровождение и манипулирующего этими данными. Преобразование таких форм в формы, пригодные для машинной обработки, — задача машин, а не людей.

Другим важным преимуществом высокоуровневых, более декларативных форм записи является то, что они лучше приспособлены к проверке во время компиляции. Для процедурных форм записи характерно сложное поведение во время выполнения программ, которое трудно анализировать на этапе компиляции. Декларативные нотации предоставляют гораздо больше возможностей для обнаружения ошибок, поскольку позволяют полнее понять запланированное поведение. Генри Спенсер.

Это понимание теоретически обосновывает набор практических приемов, которые всегда были важной частью инструментария Unix-программиста — языки очень высокого уровня, программы, управляемые данными, генераторы кода и узкоспециальные мини-языки. Объединяет их то, что все они являются способами, позволяющими поднять генерацию кода на несколько уровней выше, чтобы спецификации могли быть меньше. Ранее отмечалось, что плотность дефектов стремится к почти постоянному значению в различных языках программирования. Все указанные практические приемы означают сокращение количества строк и, соответственно, уменьшение вероятности появления ошибок.

В главе 8 описывалось использование специализированных мини-языков. В главе 14 представлены аргументы в пользу языков очень высокого уровня. В данной главе рассматривается несколько примеров конструкции программ, управляемых данными, а также ряд примеров генерации особого кода. Некоторые средства генерации кода описаны в главе 15. Как и мини-языки, данные методы позволяют радикально сократить количество строк кода в программах и соответственно уменьшить время отладки и затраты на сопровождение.

9.1. Создание программ, управляемых данными

При создании программ,

управляемых данными (data-driven programming) , код и структуры данных, на которые он воздействует, четко отделяются друг от друга и проектируются так, чтобы можно было изменять логику программы путем редактирования не кода, а структуры данных.

Создание программ, управляемых данными, иногда путают с объектно-ориентированным программированием — еще одним стилем программирования, в котором самое важное значение имеет организация данных. Между данными стилями есть как минимум два отличия. Одно из них заключается в том, что в первом случае данные представляют собой не просто состояние некоторого объекта, а фактически определяют управляющую логику программы. Тогда как основной проблемой в ОО-программировании является инкапсуляция, основная проблема при создании программ, управляемых данными, заключается в написании как можно меньшего количества фиксированного кода. Операционная система Unix имеет более устойчивые традиции в создании программ, управляемых данными, чем в ОО-программировании.

Разработку управляемых данными программ также часто путают с написанием конечных автоматов. Выразить логику конечного автомата в виде таблицы или структуры данных действительно возможно, однако запрограммированные вручную конечные автоматы обычно представляют собой жесткие блоки кода, которые гораздо труднее модифицировать, чем таблицу.

При выполнении любого вида генерации кода или создании программ, управляемых данными, необходимо помнить важное правило: решение проблемы всегда следует переносить на более высокие уровни. Не следует вручную улучшать сгенерированный код или любые промежуточные формы представления. Вместо этого следует задуматься о

способе усовершенствования или замены средства трансляции. Иначе, вероятно, выяснится, что улучшение вручную кода, который должен был быть корректно сгенерированным машиной, превратится в бесконечную трату времени.

На верхней границе шкалы сложности создание управляемых данными программ сливается с написанием интерпретаторов для p-кода или простых мини-языков, которые рассматривались в главе 8. На других границах оно сливается с генерацией кода и программированием конечных автоматов. Различия фактически не так важны. Важной частью является перемещение логики программы из жестко закодированных управляющих структур в данные.

9.1.1. Учебный пример:

ascii

Автор этой книги поддерживает программу, которая называется

ascii и представляет собой очень простую небольшую утилиту, интерпретирующую аргументы командной строки как названия ASCII-символов (ASCII, American Standard Code for Information Interchange — Американский стандартный код обмена информацией) и отображает все эквивалентные названия. Код и документация данной программы доступны на сайте проекта <http://www.catb.org/~esr/ascii>. Ниже приводится иллюстративная копия экрана.

```
esr@snark:~/WWW/writings/taoup$ ascii 10

ASCII 1/0 is decimal 016, hex 10, octal 020, bits 00010000: called ^P, DLE

Official name: Data Link Escape


ASCII 0/10 is decimal 010, hex 0a, octal 012, bits 00001010: called LF, NL

Official name: Line Feed

C escape: '\n'

Other names: Newline


ASCII 0/8 is decimal 008, hex 08, octal 010, bits 00001000: called ^H, BS

Official name: Backspace

C escape: '\b'

Other names:


ASCII 0/2 is decimal 002, hex 02, octal 002, bits 00000010: called ^B, STX

Official name: Start of Text
```

О том, что в основу данной программы положена хорошая идея, свидетельствует тот факт,

что программу можно использовать неожиданным способом — как быстрое вспомогательное CLI-средство для преобразования десятичных, шестнадцатеричных, восьмеричных и двоичных форм представления байтов.

Основная логика данной программы могла бы быть реализована в виде конструкции выбора с 128 условными переходами. Однако в таком случае код был бы громоздким и сложным в сопровождении. Кроме того, в нем смешивались бы части, которые изменяются сравнительно часто (такие как список сленговых названий символов), а также части, изменяемые редко или вообще немодифицируемые (такие как официальные названия). Помещение таких частей в один ряд условных обозначений и ошибки в ходе редактирования, вероятнее всего, затрагивали бы данные, которые должны оставаться неизменными.

Вместо этого утилита была создана как управляемая данными программа. Все строки с названиями символов находятся в табличной структуре, которая значительно крупнее любой из функций в коде (в действительности, если учитывать количество строк, она больше чем любые

три функции программы). В коде просто реализована навигация по таблице и выполнение низкоуровневых задач, таких как преобразование систем счисления. Инициализатор фактически находится в файле nametable.h, который генерируется способом, рассмотренным далее в данной главе.

Подобная организация упрощает добавление новых названий символов, изменение существующих или удаление старых названий просто путем редактирования таблицы, не затрагивая кода.

Способ организации программы является хорошим Unix-стилем, но формат ее вывода сомнительный. Трудно понять, как практически можно применить вывод в качестве ввода другой программы, поэтому утилита слабо приспособлена к взаимодействию с другими программами.

9.1.2. Учебный пример: статистическая фильтрация спама

Одним интересным случаем управляемых данными программ являются статистические самообучающиеся алгоритмы для обнаружения спама (нежелательной массы электронной почты). Целый класс программ фильтрации почты (которые легко можно найти в Web, например,

popfile, spambayes и

bogofilter) используют базу данных взаимосвязи слов как замену для сложной условной логики спам-фильтров на основе сличения с образцами.

Подобные программы стали широко распространенными в Internet очень быстро после выхода в 2002 году примечательной статьи Пола Грэхема (Paul Graham)

"A Plan for Spam" [31]. Пока наблюдался бурный рост, вызванный растущими затратами на гонку вооружений в программах сличения с образцами, идея статистической фильтрации была раньше и быстрее принята в среде Unix. Отчасти это, конечно, было связано с тем, что почти все Internet-провайдеры (которые больше всех были озабочены спамом и, следовательно, имели наибольший стимул принимать новые эффективные методики) являются Unix-предприятиями. Этому, несомненно, также способствовало согласие с

некоторыми традиционными идеями в конструировании программного обеспечения для Unix.

Традиционные спам-фильтры требуют, чтобы системный администратор (или другое ответственное лицо) поддерживал информацию об образцах текста, найденных в спаме, — имена узлов, не отправляющих ничего, кроме спама, фразы-приманки, часто используемые порнографическими сайтами или Internet-мошенниками, и аналогичные сведения. В своей статье Грэхем точно подметил, что программистам нравится идея фильтрации по образцам, и иногда они не способны "взглянуть за рамки" данного подхода, поскольку он предлагает им такие возможности проявить свою сообразительность.

С другой стороны, статистические спам-фильтры работают, накапливая информацию от пользователей о том, что те считают спамом, а что нет. Данные сведения вносятся в базы данных статистических корреляционных коэффициентов или

весов , связывающих слова или фразы с пользовательской классификацией спам/неспам. В наиболее популярных алгоритмах используются частные случаи теоремы Байеса об условных вероятностях, однако также применяются другие методики (включая различные виды полиномиального хэширования).

Во всех таких программах корреляционная проверка представляет собой сравнительно простую математическую формулу. Весовые коэффициенты, подставленные в формулу, наряду с проверяемым сообщением служат в качестве неявной управляющей структуры для фильтрующего алгоритма.

Проблема традиционных спам-фильтров на основе сличения с образцом заключается в их хрупкости. Спамеры постоянно состязаются с базами данных правил фильтрации, заставляя кураторов постоянно перенастраивать фильтры, для того чтобы "оставаться на первых позициях в гонке вооружений". Статистические спам-фильтры создают собственные правила фильтрации на основе информации пользователей.

Фактически опыт работы со статистическими фильтрами показывает, что отдельный используемый самообучающийся алгоритм гораздо менее важен, чем качество наборов данных спам/неспам, на основе которых алгоритм вычисляет весовые коэффициенты. Поэтому результаты статистических фильтров действительно больше определяются моделью данных, чем алгоритмом.

Статья

"A Plan for Spam" была ошеломляющей новостью, поскольку ее автор убедительно доказал, что простой, даже грубый статистический подход дает меньшее количество принятых за спам и не являющихся таковыми сообщений, чем могли бы предоставить любые сложные методики на основе сличения с образцом или человек, просматривающий письма. Unix-программистам проще избежать соблазна искусных методик сличения с образом, чем их коллегам в других культурах программирования, которые не так привязаны к принципу K.I.S.S.

9.1.3. Учебный пример: программирование метаклассов в

fetchmail

Утилита

fetchmailconf(1) , конфигуратор файлов профилей, поставляемая с программой

fetchmail(1) , содержит полезный пример передовой программы, управляемой данными, написанной в объектно-ориентированном языке очень высокого уровня.

В октябре 1997 года серия вопросов в списке рассылки пользователей fetchmail продемонстрировала тот факт, что конечные пользователи сталкивались с возрастающими проблемами при создании конфигурационных файлов для

fetchmail. В файле используется простой классический для Unix свободный формат синтаксиса. Однако данный файл может оказаться неприступно сложным, когда пользователь имеет учетные записи POP3 и IMAP на нескольких узлах. Несколько упрощенная версия конфигурационного файла

fetchmail приведена в примере 9.1.

Цель создания

fetchmaikonf заключалась в том, чтобы полностью скрыть синтаксис управляющего файла за стильным, эргономичным GUI-интерфейсом с кнопками выбора, бегунками и формами для заполнения. Однако в бета-версии была проблема: программа могла легко создавать конфигурационные файлы, исходя из действий, предпринятых пользователем в GUI, но не могла считывать и редактировать уже существующие файлы.Пример 9.1. Синтаксис файла fetchmailrc

```
set postmaster "esr"

set daemon 300


poll imap.ccil.org with proto IMAP and options no dns

 aka snark.thyrsus.com locke.ccil.org ccil.org

  user esr there is esr here

   options fetchall dropstatus warnings 3600


poll imap.netaxs.com with proto IMAP

 user "esr" there is esr here options dropstatus warnings 3600
```

Анализатор для синтаксиса конфигурационного файла

fetchmail является довольно сложным. Фактически он написан с использованием программ

yacc и

lex , двух классических инструментов Unix для создания кода синтаксического анализатора на языке C. Вначале разработчикам показалось, что для того чтобы с помощью

fetchmailconf можно было редактировать существующие конфигурационные файлы, понадобится воспроизвести тот же сложный синтаксический анализатор в языке реализации fetchmailconf — Python.

Такая тактика выглядела бесперспективной. Даже если не принимать во внимание объем необходимой дублирующей работы, очень трудно быть уверенным в том, что два

синтаксических анализатора, написанных на двух различных языках, допускают использование одной и той же грамматики. Обеспечить в дальнейшем их синхронизацию по мере развития конфигурационного языка будет чрезвычайно сложно. Данный подход нарушал бы правило SPOT, описанное в главе 4.

На какое-то время автор был поставлен в тупик данной проблемой. Разрешило ее интуитивное понимание того, что программа

fetchmailconf может использовать собственный синтаксический анализатор

fetchmail в качестве фильтра. В результате в

fetchmail был добавлен параметр --configdump, который позволял бы анализировать файл .fetchmailrc и отправлять результаты на стандартный вывод в формате Python-инициализатора. Для приведенного выше файла результаты выглядели бы приблизительно так, как показано в примере 9.2 (для экономии места некоторые данные, не связанные с примером, опущены).

Основное препятствие было преодолено. Интерпретатор Python мог затем оценить вывод

fetchmail --configdump и прочесть доступную для

fetchmailconf конфигурацию как значение переменой "fetchmail".

Однако описанное препятствие было не последним. Было действительно необходимо не только предоставить

fetchmail существующую конфигурацию, но превратить ее в связанное дерево действующих объектов. В данном дереве было бы 3 вида объектов: Configuration (объект верхнего уровня, представляющий всю конфигурацию), Site (представляющий один из серверов для опроса) и User (представляющий пользовательские данные, связанные с узлом). Файл в примере описывает 3 объекта Site, каждый из которых связан с одним пользовательским объектом.

Данные 3 класса объектов уже существовали в

fetchmailconf . Каждый из них имел метод, который заставлял его выводить на экран GUI-панель редактирования для модификации своего экземпляра данных. Последняя проблема сводилась к некоторому преобразованию статических данных в Python-инициализаторе в действующие объекты.Пример 9.2. Дамп Python-структуры для конфигурации

fetchmail

```
fetchmailrc = {

 'poll_interval':300,

 "logfile":None,

 "postmaster":"esr",

 'bouncemail':TRUE,

 "properties":None,

 'invisible':FALSE,

 'syslog' :FALSE,
```

```
# List of server entries begins here
# (Ниже начинается список серверов)
'servers': [
 # Entry for site `imap.ccil.org' begins:
 # (Начало записи для узла imap.ccil.org:)
 {
  "pollname":"imap.ccil.org",
  'active':TRUE,
  "via":None,
  "protocol":"IMAP",
  'port':0,
  'timeout':300,
  'dns':FALSE,
  "aka":["snark.thyrsus.com","locke.ccil.org","ccil.org"],
  'users': [
   {
    "remote":"esr",
    "password":"masked_one",
    'localnames':["esr"],
    'fetchall':TRUE,
    'keep':FALSE,
    'flush':FALSE,
    "mda":None,
    'limit':0,
    'warnings':3600,
   }
  ' ]
 }
'
 # Entry for site `imap.netaxs.com' begins:
```

```
  # (Начало записи для узла imap.netaxs.com:)
  {
   "pollname":"imap.netaxs.com",
   'active':TRUE,
   "via":None,
   "protocol":"IMAP",
   'port':0,
   'timeout':300,
   'dns':TRUE,
   "aka":None,
   'users': [
    {
     "remote":"esr",
     "password":"masked_two",
     'localnames':["esr"],
     'fetchall':FALSE,
     'keep':FALSE,
     'flush':FALSE,
     "mda":None,
     'limit':0,
     'warnings':3600,
    }
   ' ]
  }
  '
 ]
}
```

Рассматривалась идея написания связующего уровня, который имел бы явную информацию о структуре всех 3 классов и использовал бы данную информацию для просмотра инициализатора при создании соответствующих объектов. Однако данная идея была отклонена, поскольку существовала вероятность добавления со временем новых членов класса, по мере создания новых функций в конфигурационном языке. Если бы код создания

объектов был написан таким очевидным путем, то он также был бы хрупким и склонным к рассинхронизации при изменении определения классов либо структуры инициализатора, выводимой с помощью генератора отчетов --configdump. Подобный подход приводит к бесконечному появлению ошибок.

Более надежным способом было бы использование создание программы, управляемой данными, т.е. кода, который анализировал бы форму и члены инициализатора, опрашивал бы определения классов об их членах, а затем согласовывал бы оба набора.

Программисты, работающие с Lisp и Java, называют данную методику

интроспекцией (introspection) . В некоторых других объектно-ориентированных языках она называется

программированием метаклассов (metaclass hacking) и, как правило, считается "черной магией", понятной только "посвященным". В большинстве объектно- ориентированных языков данная методика не поддерживается вообще, а в тех языках, где она поддерживается (среди них Perl и Java), она часто сложна и ненадежна. Однако в языке Python средства интроспекции и программирования метаклассов исключительно доступны.

В примере 9.3 приведен фрагмент кода приблизительно со строки 1895 в версии 1.43.
Пример 9.3. Код метакласса copy_instance

```
def copy_instance(toclass, fromdict):

#Make a class object of given type from a conformant dictionary.

 class_sig = toclass.__dict__.keys(); class_sig.sort()

 dict_keys = fromdict.keys(); dict_keys.sort()

 common = set_intersection(class_sig, dict_keys)

 if 'typemap' in class_sig:

  class_sig.remove('typemap')

 if tuple(class_sig) != tuple(dict_keys):

  print "Conformability error"

#print "Class signature: " + `class_sig`

#print "Dictionary keys: " + `dict_keys`

  print "Not matched in class signature: "+ \

   `set_diff(class_sig, common)`

  print "Not matched in dictionary keys: "+ \

   `set_diff(dict_keys, common)`

  sys.exit(1)

 else:

  for x in dict_keys:
```

```
  setattr(toclass, x, fromdict[x])
```

Большую часть в примере представляет код контроля ошибок, учитывая возможность того, что члены класса и генерация отчета --configdump выпали из синхронизации. Такая проверка гарантирует, что в случае возникновения сбой в коде будет обнаружен на ранней стадии, т.е. реализуется правило исправности. Главной частью кода являются две последние строки, которые устанавливают атрибуты в классе из соответствующих членов в словаре. Данные строки эквивалентны следующим строкам.

```
def copy_instance(toclass, fromdict):

 for x in fromdict.keys():

  setattr(toclass, x, fromdict[x])
```

Если разрабатываемый код настолько же прост, то весьма вероятно, что он верен. В примере 9.4 приведен код, вызывающий данный метакласс.

Ключевым моментом в данном коде является то, что он проходит 3 уровня инициализатора (конфигурация/сервер/пользователь), устанавливая корректные объекты каждого уровня в списки, содержащиеся в следующем объекте более высокого уровня. Поскольку метакласс copy_instance управляется данными и является полностью общим, его можно использовать во всех 3 уровнях для 3 различных типов объектов.

Данный пример — пример новой школы. Язык Python был создан гораздо позднее 1990 года. Однако пример отражает идеи, которые возвращаются к Unix-традициям 1969 года. Если бы размышления над Unix-программированием, практикуемым предшественниками, не научили бы автора "конструктивной лени" — настаивая на повторном использовании кода и отказе от написания дублирующегося связующего кода в соответствии с правилом SPOT — он мог бы "удариться" в программирование синтаксического анализатора на языке Python. Главное понимание того, что сама программа

fetchmail могла бы превратиться в синтаксический анализатор конфигурации

fetchmailconf , возможно, никогда бы не пришло.Пример 9.4. Вызывающий контекст для copy_instance

```
#Сложная часть - инициализация объектов из глобального класса

#`configuration'.

#`Configuration' - верхний уровень объектного дерева,

#который планируется изменить

Configuration = Controls()

copy_instance(Configuration, configuration)

Configuration.servers = [];

for server in configuration['servers']:

 Newsite = Server()

 copy_instance(Newsite, server)

 Configuration.servers.append(Newsite)
```

```
 Newsite.users = [];

 for user in server['users']:

  Newuser = User()

  copy_instance(Newuser, user)

  Newsite.users.append(Newuser)
```

Другое понимание (того, что метакласс copy_instance может быть общим) происходит из Unix-традиции старательного поиска способов избежать кодирования вручную. Но особенно, Unix-программисты привыкли к написанию спецификаций для генерации синтаксических анализаторов для обработки языков разметки. Это скоро привело к предположению, что остальная часть работы может быть выполнена путем некоторого общего обхода дерева конфигурационной структуры. Для четкого разрешения задачи проектирования необходимо было два отдельных (один над другим) этапа создания программы, управляемой данными.

Интуитивное понимание подобное описанному может быть необыкновенно действенным. Рассматриваемый код был написан в течение приблизительно 90 минут, был работоспособен при первом запуске и с тех пор в течение многих лет оставался стабильным (единственный сбой произошел при обработке исключительной ситуации в присутствии действительного перекоса версий). Данный код содержит менее 40 строк и великолепно прост. Не существует способа, при котором примитивный подход полного создания второго синтаксического анализатора мог бы привести к созданию такого же удобства сопровождения, такой же надежности или компактности. Повторное использование кода, упрощение, обобщение, ортогональность — Дзэн операционной системы Unix в действии.

В главе 10 рассматривается синтаксис файла конфигурации

fetchmail в качестве примера стандартного shell-подобного метаформата для конфигурационных файлов. В главе 14

fetchmailconf используется как пример, демонстрирующий мощность языка Python в быстрой разработке GUI-интерфейсов.

9.2. Генерация специального кода

Операционная система Unix оснащена несколькими мощными генераторами кода специального назначения, предназначенного для таких целей, как создание лексических анализаторов (tokenizers) и синтаксических анализаторов; они рассматриваются в главе 15. Однако существуют более простые, легковесные виды генераторов кода, которые можно использовать для облегчения работы программиста и не требуют знания теории компиляторов или написания процедурной логики (подверженной ошибкам).

Ниже приводится два иллюстративных учебных примера.

9.2.1. Учебный пример: генерация кода для

ascii -дисплеев

Запущенная без аргументов программа

ascii генерирует экран справочной информации по использованию, подобный распечатке, приведенной в примере 9.5.

Данный экран достаточно тщательно спроектирован, чтобы уместиться в 23 строки и 79 колонок, поэтому он помещается в терминальном окне 24x80.

Таблицу можно было бы создавать во время выполнения, т.е. на лету. Подгонка десятичных и шестнадцатеричных столбцов была бы достаточно простой. Однако существует достаточно необычных, делающих код крайне неудобным, случаев от перенесения строк таблицы в нужных местах до вывода таких неотображаемых символов, как NUL, вместо обычных символов. Более того, столбцы понадобилось бы неравномерно заполнять пробелами, чтобы заставить таблицу уместиться в 79 колонках. Но любой Unix-программист автоматически выражал бы таблицу как блок данных, прежде чем обнаружил бы данные проблемы.

Самым примитивным способом создания справочного экрана было бы помещение каждой строки в C-инициализатор в исходном коде ascii.c, а затем заставить код, проходящий через инициализатор, выписывать строки. Проблема такого метода заключается в том, что дополнительные данные в формате C-инициализатора (завершающие разделители строк, строковые кавычки, запятые) удлиняли бы строки более 79 символов. Это привело бы к переносу строк и усложнило бы преобразование внешнего вида кода к внешнему виду вывода, что, в свою очередь, усложнило бы редактирование справочного дисплея, который и без этого сложно было уместить в экран на 24x80 растровых ячеек.

Более сложный метод использования режима вставки строк в препроцессоре ANSI C приводит к другому варианту той же проблемы. По существу, любой способ явного включения в код справочного экрана задействовал бы пунктуацию в начале и конце строки, для которой не было места[94]. А копирование на экран таблицы из файла во время выполнения выглядело ненадежно. В конце концов, файл мог быть утерян. Пример 9.5. Справка по использованию программы

ascii

Usage: ascii [-dxohv] [-t] [char-alias...]

-t = one-line output -d = Decimal table -o = octal table -x = hex table

-h = This help screen -v = version information

Prints all aliases of an ASCII character. Args may be chars, C \-escapes,

English names, ^-escapes, ASCII mnemonics, or numerics in decimal/octal/hex.

Dec Hex Dec Hex Dec Hex Dec Hex Dec Hex Dec Hex Dec Hex Dec Hex

0 00 NUL 16 10 DLE 32 20 48 30 0 64 40 @ 80 50 P 96 60 ` 112 70 p

1 01 SOH 17 11 DC1 33 21 ! 49 31 1 65 41 A 81 51 Q 97 61 a 113 71 q

2 02 STX 18 12 DC2 34 22 " 50 32 2 66 42 B 82 52 R 98 62 b 114 72 r

3 03 ETX 19 13 DC3 35 23 # 51 33 3 67 43 C 83 53 S 99 63 c 115 73 s

4 04 EOT 20 14 DC4 36 24 $ 52 34 4 68 44 D 84 54 T 100 64 d 116 74 t

5 05 ENQ 21 15 NAK 37 25 % 53 35 5 69 45 E 85 55 U 101 65 e 117 75 u

6 06 ACK 22 16 SYN 38 26 & 54 36 6 70 46 F 86 56 V 102 66 f 118 76 v

7 07 BEL 23 17 ETB 39 27 ' 55 37 7 71 47 G 87 57 W 103 67 g 119 77 w

8 08 BS 24 18 CAN 40 28 (56 38 8 72 48 H 88 58 X 104 68 h 120 78 x

9 09 HT 25 19 EM 41 29) 57 39 9 73 49 I 89 59 Y 105 69 i 121 79 y

10 0A LF 26 1A SUB 42 2A * 58 3A : 74 4A J 90 5A Z 106 6A j 122 7A z

11 0B VT 27 1B ESC 43 2B + 59 3B ; 75 4B K 91 5B [107 6B k 123 7B {

12 0C FF 28 1C FS 44 2C , 60 3C < 76 4C L 92 5C \ 108 6C l 124 7C |

13 0D CR 29 1D GS 45 2D - 61 3D = 77 4D M 93 5D] 109 6D m 125 7D }

14 0E SO 30 1E RS 46 2E . 62 3E > 78 4E N 94 5E ^ 110 6E n 126 7E ~

15 0F SI 31 1F US 47 2F / 63 3F ? 79 4F O 95 5F _ 111 6F o 127 7F del

Данная проблема решается следующим образом. В состав исходного дистрибутива входит файл с именем splashscreen, в котором содержится такой же экран использования, как в приведенном примере. Исходный код на C содержит следующую функцию.

```
void showHelp(FILE *out, char *progname) {
 fprintf(out,"Usage: %s [-dxohv] [-t] [char-alias...]\n", progname);
#include "splashscreen.h"
 exit(0);
}
```

Файл splashscreen.h генерируется инструкцией make-файла.

```
splashscreen.h: splashscreen
 sed &amp;lt;splashscreen &amp;gt;splashscreen.h \
  -e 's/\\/\\\\/g' -e 's/"/\\"/' -e 's/.*/puts("&amp;amp;");/'
```

Поэтому при сборке программы файл splashscreen автоматически преобразовывается в серию вызовов функции вывода, которые C-препроцессор затем включает в необходимую функцию.

Путем создания кода из данных поддерживается редактируемая версия справочного экрана, идентичного его представлению на дисплее. В результате этого повышается прозрачность. Более того, при желании можно модифицировать справочный экран, не затрагивая C-кода вообще, а верное внешнее представление будет автоматически получено при следующей сборке.

По тем же причинам инициализатор, содержащий строки синонимичных названий, также генерируется посредством sed-сценария в make-файле из файла с именем nametable, входящего в состав исходного дистрибутива

ascii. Большая часть файла nametable просто копируется в C-инициализатор. Но процесс генерации упростил бы адаптацию данного средства для других 8-битовых наборов символов, таких как ISO-8859 (Latin-1 и подобные).

Данный пример почти тривиален, однако он, тем не менее, иллюстрирует преимущества генерации как простого, так и особого кода. Подобные методики могли бы применяться для более крупных программ, предоставляя, соответственно, еще большие преимущества.

9.2.2. Учебный пример: генерация HTML-кода для табличного списка

Предположим, что требуется поместить страницу табличных данных на Web-страницу. Необходимо, чтобы первые несколько строк выглядели, как в примере 9.6. Пример 9.6. Необходимый формат вывода для таблицы звезд

Aalat David Weber The Armageddon Inheritance

Aelmos Alan Dean Foster The Man who Used the Universe

Aedryr Steve Miller/Sharon Lee Scout's Progress

Aergistal Gerard Klein The Overlords of War

Afdiar L. Neil Smith Tom Paine Maru

Agandar Donald Kingsbury Psychohistorical Crisis

Aghirnamirr Jo Clayton Shadowkill

Примитивнейший способ решения данной задачи заключался бы в написании вручную HTML-кода для необходимого внешнего представления. Таким образом, каждый раз, когда потребуется добавить новое имя, придется вручную писать еще один набор тегов <tr> и <td> для новой записи. Такая необходимость очень быстро стала бы утомительной. Но еще хуже то, что при изменении формата списка каждую запись потребуется кодировать вручную.

Внешне разумный способ решить данную проблему заключался бы в том, чтобы внести данные в трехстолбцовую таблицу в базе данных, а затем использовать некоторую причудливую CGI-методику[95] или поддерживающий базы данных шаблонный процессор, например, PHP для создания страницы на лету. Однако, предположим, разработчику известно, что список не будет изменяться очень часто, и не требуется запускать сервер баз данных для того, чтобы отображать данные, а также нежелательно загружать сервер излишним CGI-трафиком.

Существует простое решение: поместить данные в файл в простом табличном формате, см. пример 9.7. Пример 9.7. Модель таблицы звезд

Aalat :David Weber :The Armageddon Inheritance

Aelmos :Alan Dean Foster :The Man who Used the Universe

Aedryr :Steve Miller/Sharon Lee :Scout's Progress

Aergistal :Gerard Klein :The Overlords of War

Afdiar :L. Neil Smith :Tom Paine Maru

Agandar :Donald Kingsbury :Psychohistorical Crisis

Aghirnamirr :Jo Clayton :Shadowkill

Можно было бы обойтись без явного использования двоеточия в качестве разделителя полей, используя модель, содержащую в качестве разделителя два или более пробела. Однако явный разделитель предохраняет от ошибок, связанных со случайным двойным нажатием пробела в процессе редактирования значений полей.

Затем создается сценарий в shell, Perl, Python или Tcl, который преобразовывает данный файл в HTML-таблицу. Сценарий запускается каждый раз после добавления новой записи. Способ старой школы Unix был бы связан с почти нечитаемым вызовом

sed(1)

sed -e 's,^,<tr><td>,' -e 's,$,</td></tr>,' -e 's,$,</td><td>,g'

или, возможно, с несколько более очевидной

awk(1) -программой.

awk -F: ' {printf
("<tr><td>%s</td><td>%s</td><td>%s</td></tr>\n", \

$1, $2, $3)}'

Если какой-либо из данных примеров вызывает интерес, но непонятен, то следует прочесть документацию на программу

sed(1) или

awk(1) . В главе 8 отмечалось, что вторая утилита почти совсем вышла из употребления. Первый редактор

sed до сих пор остается важным Unix-инструментом, который подробно не рассматривался ввиду того, что (a) Unix-программисты уже знают о нем и (b) тем, кто не является Unix-программистом, нетрудно будет узнать о его использовании из документации, как только они поймут основные идеи конвейеров и перенаправления.

Решение новой школы может основываться на приведенном ниже коде на Python или эквивалентном коде на Perl.

for row in map(lambda x:x.rstrip().split(':'),sys.stdin.readlines()):

print "<tr><td>" + "</td><td>".join(row) + "</td></tr>"

Написание и отладка данных сценариев занимает примерно 5 минут, несомненно, меньше, чем потребовалось бы для создания первоначального HTML-кода вручную или создания и проверки базы данных. Комбинация таблицы и данного кода будет гораздо проще в сопровождении, чем недопроектированный созданный вручную HTML-код или перепроектированная база данных.

Другое преимущество данного способа решения проблемы заключается в том, что поиск информации и редактирование главного файла несложно осуществлять с помощью простого текстового редактора. Еще одним преимуществом является то, что можно экспериментировать с различными преобразованиями таблицы в HTML путем тонкой

настройки генерирующего сценария или просто создать подмножество отчета, поместив перед ним фильтр

grep(1) .

Автор этой книги в настоящее время использует данную технику для сопровождения Web-страницы, на которой перечислены тестовые

fetchmail- узлы. Приведенный выше пример является вымышленным только потому, что публикация реальных данных открыла бы учетные записи и пароли пользователей.

Данный пример был отчасти менее тривиальным, чем предыдущий. В данном случае был спроектирован способ разделения между содержанием и форматированием, а также генерирующий сценарий, функционирующий как таблица стилей. (Это еще один способ разделения механизма и политики.)

Урок во всех случаях один и тот же — возможность выполнять как можно меньше работы. Позвольте данным формировать код. Опирайтесь на имеющиеся средства. Отделяйте механизм от политики. Опытные Unix-программисты учатся видеть подобные возможности быстро и автоматически. "Конструктивная лень" является одним из важнейших качеств высококлассного программиста.

10

Конфигурация: правильное начало

Стоит внимательно посмотреть на наши истоки, и результаты организуются сами собой.
—Александр Кларк (Alexander Clark)

В операционной системе Unix программы могут обмениваться данными со своим окружением различными способами. Эти способы удобно разделить на (a) опрос параметров среды и (b) интерактивные каналы. В данной главе основное внимание уделено опросу параметров среды. Интерактивные каналы рассматриваются в следующей главе.

10.1. Конфигурируемые параметры

Прежде чем погрузиться в детали различных видов конфигурации программ, следует ответить на важнейший вопрос: "Какие параметры должны быть конфигурируемыми?"

Интуитивно Unix-программист ответит: "Все". Правило разделения, которое рассматривалось в главе 1, подталкивает Unix-программистов создавать механизм и там, где это возможно, передавать решения о политике на ближайшие к пользователю уровни. Хотя в таком случае часто создаются мощные и полезные для высококвалифицированных пользователей программы, столь же часто создаются интерфейсы, ошеломляющие новичков и случайных пользователей избытком возможных вариантов и конфигурационных файлов.

В ближайшее время Unix-программисты не собираются избавляться от склонности

разрабатывать программы для коллег или наиболее опытных пользователей (в главе 20 рассматривается вопрос о том, будут ли такие перемены действительно желаемыми). Поэтому, вероятно, полезнее будет изменить вопрос: какие параметры

не должны быть конфигурируемыми? В Unix-практике имеются некоторые определяющие принципы для решения данной проблемы.

Прежде всего,

не следует создавать конфигурационные ключи для тех параметров, которые можно надежно определить автоматическим путем. Такая ошибка поразительно широко распространена. Вместо этого, рекомендуется искать способы устранения конфигурационных ключей путем автоопределения или пытаться использовать альтернативные методы до тех пор, пока один из них не достигнет цели. Если разработчик считает такой подход грубым или слишком дорогим, то ему следует спросить себя, не начал ли он преждевременную оптимизацию.

Я столкнулся с одним из наилучших примеров автоопределения, когда мы с Деннисом Ритчи переносили Unix на Interdata 8/32. Это была машина с обратным порядком следования байтов. Требовалось генерировать для нее данные на компьютере PDP-11, записывать магнитную ленту и загружать эту ленту в Interdata. Обычно ошибка возникала, когда мы забывали изменить порядок следования байтов. Ошибка контрольной суммы показывала, что необходимо отключить ленту, снова подключить ее к PDP-11, заново создать ленту, отключить ее и подключить опять. Однажды Деннис усовершенствовал программу считывания магнитной ленты на Interdata таким образом, что в случае возникновения ошибки контрольной суммы лента перематывалась, активизировался ключ "byte flip" (преобразование порядка байтов) и программа повторно считывала данные с ленты. Вторая ошибка контрольной суммы прерывала загрузку, но в 99% случаев осуществлялось повторное считывание ленты и выполнение верных действий. Продуктивность нашей работы мгновенно возросла, и с того времени мы почти полностью игнорировали порядок следования байтов на ленте. Стив Джонсон.

Существует хорошее эмпирическое правило: необходимо приспосабливаться, если затраты на это не превышают 0,7 сек задержки. 0,7 сек — "магическое" число, поскольку, как обнаружил Джеф Раскин во время разработки Canon Cat, люди почти не способны заметить более короткую задержку при запуске. Она теряется среди ментальных издержек при переключении фокуса внимания.

Второй принцип заключается в том, что

ключи оптимизации должны быть невидимыми для пользователей . Экономичная работа программы — задача

проектировщика , а не пользователя. Минимальный прирост производительности, который может получить пользователь с помощью ключей оптимизации, обычно не стоит затрат, связанных со сложностью интерфейса.

Unix всегда избегала бессмыслицы файловых форматов (таких как длина записи, фактор блокировки и другие), однако подобная бессмыслица вернулась в чрезмерной, вязкой массе конфигурации. Принцип KISS стал принципом MICAHI: make it complicated and hide it (усложнить и скрыть). Дуг Макилрой.

Наконец,

с помощью конфигурационного ключа не следует делать то, что можно сделать при помощи сценария-упаковщика или тривиального конвейера . Не следует вносить в программу дополнительную сложность, если для решения задачи можно без труда задействовать другие

программы. (В главе 7 показано, почему утилита

ls(1) не содержит встроенного средства формирования страниц или параметра для вызова такого средства.)

Существует несколько более общих вопросов, которые следует обдумывать всякий раз при появлении желания добавить конфигурационный параметр.

• Можно ли исключить данную функцию? Почему "раздувается" справочное руководство, а пользователь перегружен излишними подробностями?

• Можно ли безопасно изменить обычный режим работы программы так, чтобы данный параметр был необязательным?

• Является ли данный параметр только косметическим? Возможно, следует меньше думать о конфигурируемости пользовательского интерфейса и больше внимания уделять корректности его работы?

• Не следует ли сделать режим, активизированный данным параметром, в виде отдельной программы?

Увеличение числа излишних параметров чревато множеством негативных последствий. Одно из наименее очевидных и самых серьезных — это влияние на тестовое покрытие.

Добавление конфигурационного параметра on/off, в случае если оно не сделано очень тщательно, может привести к необходимости удвоения количества тестов. Поскольку на практике количество тестов никогда не удваивается, практический эффект заключается в сокращении количества тестов для любой заданной конфигурации. Десять параметров приводят к 1024 тестированиям, и довольно скоро возникнут реальные проблемы надежности Стив Джонсон.

10.2. Месторасположение конфигурационной информации

Классическая Unix-программа может искать управляющую информацию в 5 группах параметров начальной загрузки:

• файлы конфигурации в каталоге /etc (или в другом постоянном каталоге системы);

• системные переменные окружения;

• файлы конфигурации (или файлы профилей (dotfile)) в начальном каталоге пользователя (описание данной важной концепции приведено в главе 3);

• пользовательские переменные окружения;

• ключи и аргументы, переданные программе в командной строке во время вызова.

Данные запросы обычно выполняются в описанном выше порядке. Поэтому более поздние (локальные) установки заменяют более ранние (глобальные). Установки, найденные раньше, могут способствовать программе в определении области для дальнейшего поиска конфигурационных данных.

При выборе механизма для передачи программе конфигурационных данных необходимо помнить о том, что хорошая Unix-практика требует использования того механизма, который

наиболее близко соответствует ожидаемому времени жизни настроек. Следовательно: для настроек, которые, весьма вероятно, изменяются между вызовами программы, следует использовать ключи командной строки. Для предпочтений, которые изменяются редко, но должны находиться под индивидуальным контролем пользователя, следует использовать конфигурационный файл в начальном каталоге данного пользователя. Для хранения информации о настройках, которые должны устанавливаться системным администратором для всего узла и

не изменяются пользователями, используется конфигурационный файл в системной области.

Ниже каждая область конфигурационных параметров рассматривается более подробно, после чего приводится несколько учебных примеров.

10.3. Файлы конфигурации

Файл конфигурации (run-control file) — файл объявлений или команд, связанных с программой, которая интерпретирует его во время запуска. Если программа имеет специфическую для данного узла конфигурацию, которая совместно используется всеми пользователями узла, то часто файл конфигурации находится в каталоге /etc. (В некоторых Unix-системах для хранения таких данных имеется подкаталог /etc/conf.)

Сведения пользовательской конфигурации часто находятся в скрытом конфигурационном файле в начальном каталоге данного пользователя. Такие файлы часто называются "dotfiles" (файлы профилей), поскольку они используют Unix-соглашение о том, что имя файла, начинающееся с точки, невидимо для средств создания перечня файлов каталога[96].

Программы также могут иметь конфигурационные или скрытые каталоги. В таких каталогах собираются группы конфигурационных файлов, которые относятся к программе, но их удобнее интерпретировать отдельно (возможно потому, что они связаны с различными подсистемами программы или имеют разный синтаксис).

В настоящее время, независимо от того, используется файл или каталог, соглашение требует, чтобы месторасположение конфигурационной информации имело то же базовое имя что и считывающий ее исполняемый файл. В более раннем соглашении, которое до сих пор широко распространено в системных программах, используется имя исполняемого файла с суффиксом "rc" (от "run control" — управление работой)[97]. Поэтому опытный пользователь Unix, работая с программой "seekstuff", имеющей как общесистемную, так и пользовательскую конфигурацию, ожидает найти первую в файле /etc/seekstuff, а последнюю — в файле .seekstuff в своем начальном каталоге. Вместе с тем он не удивился бы, обнаружив файлы /etc/seekstuffrc и .seekstuffrc, особенно если программа seekstuff является какой-либо системной утилитой.

В главе 5 описывался несколько отличающийся набор правил проектирования текстовых файловых форматов, а также рассматривались способы оптимизации по различным критериям способности к взаимодействию, прозрачности и экономичности транзакций. Обычно файлы конфигурации только считываются однажды во время запуска программы и не перезаписываются. Следовательно, экономия обычно не является основной проблемой. Возможность взаимодействия и прозрачность подталкивают разработчиков к использованию текстовых форматов, разработанных для чтения людьми и редактирования с помощью обычного текстового редактора.

Несмотря на то, что семантика конфигурационных файлов, несомненно, полностью зависит от программы, существует несколько широко распространенных правил проектирования, связанных с конфигурационным синтаксисом. Они будут описаны далее, а до этого следует рассмотреть важное исключение.

Если программа является интерпретатором какого-либо языка, то ожидается, что существует просто файл команд в синтаксисе данного языка, которые выполняются во время запуска программы. Данное правило важно потому, что Unix-традиции твердо поддерживают разработку всех видов программ как языков специального назначения и мини-языков. Широко известные примеры с файлами профилей такого типа включают в себя различные командные оболочки Unix и программируемый редактор

Emacs.

Основанием для данного правила проектирования является убеждение, что частные случаи — негативное явление. Поэтому, любой ключ, изменяющий поведение языка, должен устанавливаться изнутри данного языка. Если при разработке языка обнаруживается, что выразить в самом языке все его начальные установки

невозможно , то Unix-программист сказал бы, что в конструкции имеется проблема, которую, вероятнее всего, необходимо исправить, а не разрабатывать конфигурационный синтаксис частного случая.

После рассмотрения исключения следует описать правила обычного стиля конфигурационного синтаксиса. Исторически сложилось так, что моделью для них послужил синтаксис оболочек в Unix.

1.

Поддержка объясняющих комментариев, начинающихся с символа #. Синтаксис также должен игнорировать пробел перед символом #, для того чтобы обеспечить поддержку комментариев в строках, содержащих конфигурационные директивы.

2.

Предотвращение опасных различий в пустом пространстве. То есть последовательности пробелов и символов табуляции должны синтаксически интерпретироваться как один пробел. Если формат директив ориентирован на строки, то следует игнорировать завершающие пробелы и символы табуляции в строках. Метаправило заключается в том, что интерпретация файла не должна зависеть от невидимых для человеческого глаза различий.

3.

Интерпретация нескольких пустых строк и строк комментариев как одной пустой строки. Если во входном формате в качестве разделителей записей используются пустые строки, то, возможно, понадобится гарантия того, что строка комментария не завершает запись.

4.

Лексическая интерпретация файла как простой последовательности лексем, разделенных пробелами, или строк лексем. Человеку трудно изучить, запомнить и проанализировать сложные лексические правила, поэтому их следует избегать.

5. Однако следует поддерживать строковый синтаксис для лексем, содержащих пробелы. В качестве сбалансированных разделителей можно использовать одинарные или двойные кавычки. Если поддерживаются оба вида кавычек, то не следует назначать им различную семантику, что характерно для shell, поскольку это является широко известным источником

путаницы.

6. Поддержка синтаксиса обратной косой черты для внедрения в строки непечатаемых и специальных символов. Стандартным образцом в данном случае является синтаксис escape-последовательностей, поддерживаемый компиляторами C. Поэтому, например, было бы весьма неожиданно, если бы строка "a\tb" интерпретировалась бы иначе, чем символ "a", за которым следует символ табуляции, за которым в свою очередь следует символ "b".

С другой стороны, в конфигурационном синтаксисе

не следует копировать некоторые аспекты shell-синтаксиса — по крайней мере, не имея значительной и специфической причины. В данную категорию попадают "вычурные" правила использования кавычек и скобок в shell, а также специальные метасимволы shell для масок и подстановки переменных.

Целью данных соглашений является сокращение количества нововведений, с которыми приходится справляться пользователям при чтении и редактировании конфигурационного файла для новой программы. Следовательно, если понадобилось нарушить данные соглашения, то необходимо попытаться сделать необходимость нарушений очевидной, а, кроме того, документировать синтаксис с особенной тщательностью, и (что наиболее важно) проектировать так, чтобы его можно было легко понять из примеров.

Данные правила стандартного стиля только описывают соглашения о лексемах и комментариях. Имена конфигурационных файлов, их высокоуровневый синтаксис и семантическая интерпретация синтаксиса обычно отражают специфику приложения. Однако существует несколько исключений из данного правила; одно из них — файлы профилей, которые стали "широко известными" в том смысле, что обычно содержат информацию, используемую целым классом приложений. Такое совместное использование форматов конфигурационных файлов сокращает количество нововведений, в которых приходится разбираться пользователям.

Среди таких файлов наиболее "укоренившимся", вероятно, является файл .netrc. Клиентские Internet-программы, которые должны отслеживать пары узел/пароль для пользователя, обычно могут получать их из файла .netrc, если таковой существует.

10.3.1. Учебный пример: файл .netrc

Файл .netrc — хороший пример стандартных правил в действии. Пример 10.1 содержит пароли, измененные в целях защиты реального пользователя.

Следует отметить, что разобраться в данном формате не сложно, даже встретив его впервые. Он представляет собой набор троек машина/имя/пароль, каждая из которых описывает учетную запись на удаленном узле. Такая прозрачность фактически гораздо более важна, чем экономия времени путем быстрой интерпретации или экономия пространства путем использования более компактного и непонятного файлового формата. Он позволяет сэкономить гораздо более ценный ресурс —

человеческое время, поскольку увеличивает вероятность того, что человек сможет прочесть и модифицировать данный формат без изучения руководства или использования инструмента, менее знакомого, чем простой текстовый редактор.Пример 10.1. Файл .netrc

```
#FTP-доступ к моему Web-узлу
```

```
machine Unix1.netaxs.com
 login esr
 password joesatriani

#Мой главный почтовый сервер в Netaxs
machine imap.netaxs.com
 login esr
 password jeffbeck

#Дополнительный почтовый ящик IMAP в CCIL
machine imap.ccil.org
 login esr
 password marcbonilla

#Дополнительный почтовый ящик POP в CCIL
machine pop3.ccil.org
 login esr
 password ericjohnson

#Учетная запись shell в CCIL
machine locke.ccil.org
 login esr
 password stevemorse
```

Также следует отметить то, что данный формат используется для предоставления информации нескольким службам. Это является преимуществом, поскольку означает, что конфиденциальную информацию о паролях необходимо хранить только в одном месте. Формат .netrc был разработан для первоначальной клиентской FTP-программы в Unix. Данный формат используется всеми FTP-клиентами, а также распознается некоторыми telnet-клиентами, CLI-инструментом

smbclient(1) и программой

fetchmail . При разработке Internet-клиента, который должен выполнять аутентификацию паролей посредством удаленной регистрации, правило наименьшей неожиданности требует, чтобы он по умолчанию использовал содержимое файла .netrc.

10.3.2. Переносимость на другие операционные системы

Общесистемные конфигурационные файлы являются конструкторской тактикой, которую можно применять почти во всех операционных системах, однако довольно трудно найти соответствие файлам профилей в не-Unix-окружении. Критическим элементом, недостающим в большинстве операционных систем, отличных от Unix, является действительная поддержка многопользовательской работы и понятие начального каталога для каждого пользователя. В операционной системе DOS и версиях Windows вплоть до ME (включая Windows 95 и 98), например, такая идея полностью отсутствует; вся конфигурационная информация должна храниться либо в общесистемных файлах конфигурации в фиксированной области, реестре Windows, либо в том же каталоге, из которого запускается программа. В Windows NT имеется некоторое понятие о начальных каталогах пользователей (которое послужило началом пути к Windows 2000 и XP), но оно только недостаточно поддерживается инструментальными средствами системы.

10.4. Переменные окружения

Когда запускается какая-либо Unix-программа, доступная ей среда включает в себя набор связей "имя-значение" (как имена, так и значения являются строками). Некоторые из них устанавливаются пользователем вручную, другие — системой во время регистрации пользователя, либо оболочкой или эмулятором терминала (если таковой используется). В операционной системе Unix в переменных окружения хранится информация о путях поиска файлов, системных установках по умолчанию, номер процесса и идентификатор текущего пользователя, а также другие сведения о среде выполнения программ. Команда set, введенная в приглашении shell, отображает полный список определенных в текущий момент переменных оболочки.

В языках C и C++ значения данных переменных запрашиваются с помощью библиотечной функции

getenv(3) . В языках Perl и Python во время запуска инициализируются объекты словарей среды. В других языках, как правило, используется одна из двух перечисленных моделей.

10.4.1. Системные переменные окружения

Существует множество широко известных переменных окружения, значения которых программа может получить при запуске из оболочки Unix. Данные переменные (особенно HOME) часто требуется оценить

до считывания локального файла профиля.

USER

Регистрационное имя учетной записи, под которым инициируется данный сеанс (BSD-соглашение).

LOGNAME

Регистрационное имя учетной записи, под которым инициируется данный сеанс (соглашение System V).

HOME

Начальный каталог пользователя, инициализирующего данный сеанс.

COLUMNS

Количество символьных столбцов управляющего терминала или окна эмулятора терминала.

LINES

Количество символьных строк управляющего терминала или окна эмулятора терминала.

SHELL

Имя командной оболочки данного пользователя (часто используется командами, создающими подоболочку).

PATH

Список каталогов, которые просматривает оболочка при поиске исполняемых команд, соответствующих заданному имени.

TERM

Тип терминала для консоли сеанса или окна эмулятора терминала (предварительные сведения представлены в учебном примере terminfo в главе 6). TERM — переменная, специально используемая в таких программах для создания удаленных сеансов через сеть (таких как

telnet и

ssh), предполагается, что они передают и устанавливают значение данной переменной в удаленном сеансе.

(Данный список характерен для Unix-систем, однако он не является исчерпывающим.)

Переменная HOME особенно важна, поскольку многие программы используют ее для поиска файлов профилей вызывающего пользователя (другие программы вызывают некоторые функции в динамической C-библиотеке для выяснения начального каталога вызывающего пользователя).

Следует отметить то, что некоторые или все данные системные переменные окружения могут

не устанавливаться во время запуска программы, если программа запускается методом, отличным от создания подпроцесса в shell. В частности, демоны-слушатели какого-либо TCP/IP-сокета часто не имеют таких установленных переменных, а если имеют, то их значения едва ли будут полезны.

Наконец, следует отметить, что существует традиция (проиллюстрированная переменной PATH) использования двоеточия в качестве разделителя, когда переменная окружения должна содержать несколько полей, особенно если данные поля можно интерпретировать как некоторые пути поиска. Заметим, что некоторые оболочки (особенно bash и ksh)

всегда интерпретируют разделенные двоеточиями поля в переменной окружения как имена файлов. Это, в частности, означает, что символ ~ в таких полях преобразовывается в путь к начальному каталогу данного пользователя.

10.4.2. Пользовательские переменные окружения

Несмотря на то, что приложения могут свободно интерпретировать переменные окружения за пределами определенного системой набора, в настоящее время фактическое использование такой возможности является довольно необычным. Значения переменных окружения в действительности непригодны для передачи структурированной информации в программу (хотя в принципе это реализуемо посредством синтаксического анализа значений). Вместо этого в современных Unix-приложениях используются конфигурационные файлы, а также файлы профилей.

Существует, однако, несколько конструкторских моделей, в которых могут оказаться полезными переменные окружения, определяемые пользователем.

Независимые от приложения настройки, которые должны совместно использоваться большим количеством различных программ. Данный набор "стандартных" настроек изменяется исключительно медленно, поскольку множеству программ требуется опознать каждую настройку до того, как она станет полезной[98]. Ниже приводятся стандартные переменные.

EDITOR

Имя предпочтительного для пользователя редактора (часто используется командами, создающими подоболочку)[99].

MAILER

Имя предпочтительного для пользователя почтового агента (часто используется командами, создающими подоболочку).

PAGER

Имя предпочитаемой пользователем программы для просмотра простого текста.

BROWSER

Имя предпочитаемой пользователем программы для просмотра информации в Web. Данная переменная до сих пор считается новой и еще широко не распространена.

10.4.3. Когда использовать переменные окружения

Общим для пользовательских и системных переменных окружения является то обстоятельство, что в них содержатся данные, хранение которых в большом количестве конфигурационных файлов было бы утомительным. И крайне утомительным было бы изменение данной информации во всех файлах конфигурации при изменении настроек. Обычно пользователи задают значения данных переменных в своих файлах начальной конфигурации shell-сеанса.

Значение отличается в нескольких совместно использующих файл профиля контекстах, или родительский процесс должен передать информацию множеству дочерних процессов. Ожидается, что некоторые блоки конфигурационной информации будут отличаться между несколькими контекстами, в которых вызывающий пользователь совместно использовал бы общие файлы конфигурации и файлы профиля. В качестве примера системного уровня можно рассмотреть несколько shell-сеансов, открытых через окна эмулятора терминала на рабочем столе системы X. Все они считывают одни и те же файлы профилей, но могут иметь различные значения переменных COLUMNS, LINES и TERM. (Данный метод широко использовался в shell-программировании старой школы, а в make-файлах используется до сих пор.)

Значение изменяется слишком часто для файлов профилей, но не при каждом запуске. Определяемая пользователем переменная окружения может (например) использоваться для передачи расположения файловой системы или Internet-ресурса, являющегося корнем дерева файлов, с которыми должна работать программа. Так, например, система контроля версий CVS интерпретирует переменную CVSROOT. Несколько клиентских программ чтения новостей, доставляющих новости от серверов с помощью протокола NNTP, интерпретируют переменную NNTPSERVERкак расположение запрашиваемого сервера.

Уникальное для процесса переназначение параметров необходимо выразить таким образом, чтобы не требовалось изменять командную строку вызова. Определяемая пользователем переменная среды может быть полезна в ситуациях, когда, по какой-либо причине, может быть неудобно изменять файл профиля приложения или задавать параметры командной строки (возможно, ожидается, что приложение обычно будет использоваться внутри shell-упаковщика или make-файла). Особенно важным контекстом для такого использования является отладка. Например, в Linux использование переменной LD_LIBRARY_PATH, связанной с компонующим загрузчиком

ld(1) , позволяет изменить место загрузки библиотек — возможно, для выбора версий, которые выполняют проверку переполнения буфера или профилирование (profiling).

Как правило, определяемая пользователем переменная среды может быть эффективным конструкторским выбором, когда значение изменяется настолько часто, что редактирование файла профиля становится неудобным, но не обязательно при каждом вызове (всегда устанавливать расположение с помощью параметра командной строки также было бы неудобно). Как правило, значения таких переменных следует определять

после локального файла профиля и позволить им переназначать значения файла профиля.

В Unix существует одна традиционная конструкторская модель, которую не рекомендуется применять для новых программ. Иногда пользовательские переменные окружения используются как легковесная замена для выражения настроек программы в файле конфигурации. Например, старая игра

nethack(1) для того, чтобы получить пользовательские настройки, считывала переменную окружения NETHACKOPTIONS. Такой подход характерен для старой школы. Современная практика в аналогичном случае склонялась бы к синтаксическому анализу настроек из конфигурационного файла .nethack или .nethackrc.

Проблема более раннего стиля заключается в том, что отслеживание месторасположения информации о настройках становится сложнее, чем это было бы, если бы пользователь знал, что в его начальном каталоге находится конфигурационный файл программы. Переменные среды могут быть установлены в любом из нескольких конфигурационных файлов оболочки. В операционной системе Linux в их число, вероятнее всего, входят, как минимум, файлы .profile, .bash_profile и .bashrc. Данные файлы запутаны и ненадежны, поэтому, как только

издержки кода, содержащего синтаксический анализатор параметров, стали казаться менее значительными, возникла тенденция к перемещению информации о настройках из переменных окружения в файлы профилей.

10.4.4. Переносимость на другие операционные системы

Переменные среды характеризуются исключительно ограниченной переносимостью из Unix. В операционных системах Microsoft имеются переменные окружения, смоделированные наподобие переменных окружения в Unix, а переменная PATH используется, как и в Unix, для установки путей поиска исполняемых файлов, однако большинство других переменных, которые принимаются Unix-программистами как должное (такие как идентификатор процесса или текущий рабочий каталог) не поддерживаются. В других операционных системах (включая классическую MacOS), как правило, отсутствует какой-либо локальный эквивалент переменным окружения.

10.5. Параметры командной строки

Unix-традиции поощряют использование ключей командной строки для управления программами, так чтобы параметры можно было задавать из сценариев. Это особенно важно для программ, которые выполняют функции фильтров или каналов. Существует 3 соглашения, которые определяют способ отделения параметров командной строки от обычных аргументов: исходный стиль Unix, стиль GNU и стиль инструментария X.

В оригинальной традиции Unix для обозначения параметров командной строки служат отдельные буквы с предшествующим дефисом. Параметры режимов, которые не принимают последующих аргументов, можно группировать, поэтому если -a и -b являются параметрами режима, то использование последовательностей -ab или -ba также является корректным и активизирует оба режима. Аргумент для параметра, если он существует, указывается после параметра (и отделяется от него пробелом), В данном стиле использование нижнего регистра для обозначения параметров более предпочтительно. Использование в разрабатываемой программе обозначений в верхнем регистре хорошо в том случае, если они представляют собой специальные варианты параметров, обозначаемых буквами нижнего регистра.

Исходный стиль Unix развивался на медленных телетайпах ASR-33, в которых краткость ввода была достоинством; отсюда и использование однобуквенных параметров. Удержание клавиши shift требовало определенных усилий, отсюда предпочтение для нижнего регистра и использование для активизации параметров символа "-" (вместо знака "+", возможно, более логичного в данном случае).

В GNU-стиле для обозначения параметров используются ключевые слова (вместо отдельных букв), которым предшествуют два дефиса. Данный стиль развивался гораздо позднее, когда некоторые из довольно сложных GNU-утилит начали сталкиваться с нехваткой однобуквенных ключей (это было "лечением симптома", но не избавляло от лежащей в основе проблемы). Данный стиль остается популярным, поскольку GNU-параметры читать проще, чем "алфавитную смесь" более ранних стилей. Параметры в GNU-стиле не могут быть сгруппированы без разделяющего пробела. Аргумент параметра (если есть) может отделяться либо пробелом, либо одиночным знаком равенства ("=").

Двойной дефис в GNU-стиле был выбран для того, чтобы в одной командной строке можно было недвусмысленно смешивать традиционные однобуквенные параметры и ключевые слова GNU-стиля. Таким образом, если в первоначально разрабатываемой конструкции присутствует только несколько простых параметров, то можно использовать Unix-стиль, не опасаясь "дня флага", в случае если потребуется в дальнейшем перейти на GNU-стиль. С другой стороны, если используется GNU-стиль, то хорошей практикой будет поддержка однобуквенных эквивалентов, по крайней мере, для наиболее широко используемых параметров.

В стиле X-инструментария для обозначения параметра несколько запутанно используется одиночный дефис и ключевое слово. Данный стиль интерпретируется X-инструментариями, которые фильтруют и обрабатывают определенные параметры (такие как -geometry и -display) до передачи преобразованной командной строки логике приложения для интерпретации. Стиль X-инструментария не вполне совместим с классическим стилем Unix или GNU-стилем, и его не следует использовать в новых программах, если совместимость с более ранними X-соглашениями не очень важна.

Многие инструментальные средства принимают один дефис, не связанный с какой-либо буквой параметра, как имя псевдофайла, заставляющее приложение считывать данные со стандартного ввода. Кроме того, двойной дефис традиционно распознается как сигнал прекратить интерпретацию параметров и начать буквальную интерпретацию всех последующих аргументов.

Большинство языков программирования в Unix предоставляют библиотеки, которые производят синтаксический анализ командной строки либо в классическом, либо в GNU-стиле (также интерпретируя соглашение по двойному дефису).

10.5.1. Параметры командной строки от -a до -z

Со временем часто используемые параметры в широко известных Unix-программах создали неформальный стандарт семантики для ожидаемого значения различных флагов. Ниже приводится перечень параметров и их значений, которые будут удобными и привычными для опытных пользователей операционной системы Unix.

-a

Все (без аргументов). В GNU-стиле параметр -all. Было бы очень неожиданно встретить другое значение для -a, кроме синонима --all. Примеры:

fuser(1), fetchmail(1) .

Добавить в конец, как в утилите

tar(1) . Часто образовывает пару с ключом -d для удаления.

-b

Размер буфера или блока (с аргументом). Устанавливает размер буфера или размер блока (в программе, осуществляющей архивирование или управление устройствами хранения информации). Примеры:

du(1), df(1), tar(1) .

Неинтерактивный (пакетный (batch)) режим. Если программа изначально интерактивна, то ключ -b может использоваться для подавления подсказок или установки других параметров, необходимых для принятия входных данных из файла, а не от пользователя. Пример:

flex(1) .

-c

Команда (с аргументом). Если программа является интерпретатором, обычно принимающим команды со стандартного ввода, то ожидается, что аргумент -c будет передан программе как одна строка ввода. Данное соглашение особенно четко соблюдается в оболочках и shell-подобных интерпретаторах. Примеры

: sh(1), ash(1), bsh(1), ksh(1), python(1) . Сравните с ключом -e ниже.

Проверка (check) (без аргумента). Проверяется корректность аргумента (или аргументов) файла для команды, но обычная обработка фактически не выполняется. Часто используется как параметр проверки синтаксиса в программах, которые действительно интерпретируют командные файлы. Примеры:

getty(1), perl(1) .

-d

Отладка (debug) (с аргументом или без). Устанавливает уровень отладочных сообщений. Данное значение весьма распространено.

Иногда ключ -d имеет значение "delete" (удалить) или "directory" (каталог).

Определить (define) (с аргументом). Устанавливает значение какого-либо символа в интерпретаторе, компиляторе или (особенно) в приложении с функциями макропроцессора. Моделью послужило использование ключа -D в препроцессоре макрокоманд компилятора C. Это устойчивая ассоциация для большинства Unix-программистов; ее не следует нарушать.

-e

Выполнить (execute) (с аргументом). Программы, которые являются упаковщиками или могут использоваться как упаковщики, часто допускают ключ -e для определения программы, которой они передают управление.

Редактирование (edit). Программа, способная открыть ресурс либо в режиме только для чтения, либо в режиме редактирования, может принимать ключ -e для активизации режима редактирования. Примеры:

crontab(1) и утилита

get(1) системы контроля версий SCCS.

Иногда ключ -e имеет значение "exclude" (исключить) или "expression" (выражение).

-f

Файл (с аргументом). Очень часто используется с аргументом для указания файла входных (или реже выходных) данных для программ, которым требуется произвольный доступ к входным или выходным данным (там, где недостаточно перенаправления посредством символов < или >). Классическим примером является утилита

tar(1) ; существует также множество других примеров. Кроме того, данный ключ часто

используется для указания того, что аргумент, обычно принимаемый из командной строки, необходимо взять из файла. Классические примеры:

awk(1) и

egrep(1) . Сравните с ключом -o ниже. Часто -f представляет собой входной аналог -o.

Форсировать (force) (обычно без аргумента). Форсирует некоторые операции (такие как блокировка или разблокировка файлов), которые обычно осуществляются по условию. Данное значение менее распространено.

Демоны часто используют ключ -f, комбинируя оба значения, для того чтобы форсировать обработку конфигурационного файла, расположенного в нестандартном месте. Примеры:

ssh(1), httpd(1) и многие другие демоны.

-h

Заголовки (headers) (обычно без аргумента). Включает, подавляет или модифицирует заголовки табличных отчетов, сгенерированных программой. Примеры:

pr(1), ps(1) .

Помощь (help). В настоящее время данное значение распространено меньше, чем можно ожидать. В большинстве случаев разработчики ранней Unix считали справку лишней тратой памяти, которую они не могли себе позволить. Вместо этого они создавали страницы руководства (что определенным способом сформировало соответствующий стиль справочных руководств в Unix, см. главу 18).

-i

Инициализировать (initialize) (обычно без аргумента). Устанавливает некоторый критический ресурс или базу данных, связанную с программой, в первоначальное или пустое состояние. Пример:

ci(1) в RCS.

Диалоговый (интерактивный (interactive)) режим (обычно без аргумента). Заставляет программу (которая обычно этого не делает) запрашивать подтверждение выполняемых операций. Существуют классические примеры (

rm(1), mv(1)), однако данное значение не является общепринятым.

Включить (include) (с аргументом). Добавляет имя файла или каталога в список просматриваемых приложением ресурсов. Все Unix-компиляторы с любым эквивалентом включения файла исходного кода в своих языках используют ключ -I именно в этом смысле. Было бы крайне неожиданно увидеть иное использование данного параметра.

-k

Хранить (keep) (с аргументом). Подавляет обычное удаление какого-либо файла, сообщения или ресурса. Примеры:

passwd(1), bzip(1) и

fetchmail(1) .

Иногда ключ -k имеет значение "kill" (уничтожить, завершить).

-l

Вывести список (list) (без аргумента). Если разрабатываемая программа является архиватором или интерпретатором/программой просмотра для какого- либо каталога или архива, то было бы очень неожиданно использовать ключ -l для действий, отличных от запроса списка объектов. Примеры:

arc(1), binhex(1), unzip(1) . (Утилиты

tar(1) и

cpio(1) являются исключениями.)

В программах, которые уже являются генераторами отчетов, ключ -l почти без исключений означает "long" (длинный) и отображает список в длинном формате, где открываются более подробные сведения по сравнению со стандартным режимом. Примеры:

ls(1), ps(1) .

Загрузить (load) (с аргументом). Если программа является компоновщиком или интерпретатором языка, то ключ -l неизменно загружает библиотеку в некотором соответствующем состоянии. Примеры:

gcc(1), f77(1), emacs(1) .

Регистрация в системе (login). В таких программах, как

rlogin(1) и

ssh(1) , в которых необходимо указать сеть, это можно сделать с помощью ключа -l.

Иногда -l имеет значение "length" (длина, длительность) или "lock" (блокировка).

-m

Сообщение (message) (с аргументом). Ключ -m передает свой аргумент в программу как строку сообщения для протоколирования или извещения. Примеры:

ci(1), cvs(1) .

Иногда ключ -m имеет значение "mail" (почта), "mode" (режим) или "modification-time" (время изменения).

-n

Число (number) (с аргументом). Используется, например, для указания диапазонов номеров страниц в таких программах, как

head(1), tail(1), nroff(1) и

troff(1) . Некоторые сетевые инструменты, обычно отображающие DNS-имена, принимают ключ -n как параметр, который вызывает отображение IP-адресов вместо имен. Основные примеры:

ifconfig(1) и

tcpdump(1) .

Нет (not) (без аргумента). Используется для подавления обычных действий в таких программах, как

make(1) .

-o

Вывод (output) (с аргументами). Используется для того, чтобы указать программе в командной строке выходной файл или устройство по имени. Примеры:

as(1), cc(1), sort(1) . Во всех программах с интерфейсами, подобными интерфейсу компилятора, было бы крайне неожиданно увидеть иное использование данного параметра. Программы, поддерживающие ключ -o часто (как gcc) имеют логику, которая позволяет опознавать его как после обычных аргументов, так и перед ними.

-p

Порт (port) (с аргументом). Главным образом применяется для параметров, которые определяют номера TCP/IP-портов. Примеры:

cvs(1) , инструменты PostgreSQL,

smbclient(1), snmpd(1), ssh(1) .

Протокол (protocol) (с аргументом). Примеры:

fetchmail(1), snmpnetstat(1) .

-q

Подавление вывода (quiet) (обычно без аргумента). Подавляет обычное отображение результата или сведений диагностики. Данное значение является весьма распространенным. Примеры:

ci(1), co(1), make(1) . См. также значение "подавление вывода" ключа -s.

-r (а также -R)

Рекурсия (recurse) (без аргумента). Если программа выполняет операции над каталогом, данный параметр может указывать на то, что операции рекурсивно повторяются для всех входящих в данный каталог подкаталогов. Любое другое применение данного ключа в утилите, обрабатывающей каталоги, было бы весьма неожиданным. Классическим примером, несомненно, является утилита

cp(1) .

Реверс (reverse) (без аргумента). Примеры:

ls(1), sort(1) . Данный параметр может использоваться в фильтре для реверсирования своего обычного действия преобразования (сравните с ключом -d).

-s

Подавление вывода (silent) (без аргументов). Подавляет обычное отображение сведений диагностики или результатов (аналогичен ключу -q; когда поддерживаются оба ключа, то q означает "quiet" (тихо), а -s — "utterly silent" (очень тихо)). Примеры:

csplit(1), ex(1), fetchmail(1) .

Тема сообщения (subject) (с аргументом).

Всегда используется в данном значении в командах, которые отправляют или манипулируют почтовыми сообщениями или новостями. Крайне важно обеспечить поддержку данного параметра, поскольку он ожидается программами, отправляющими почту. Примеры:

mail(1), elm(1), mutt(1) .

Иногда ключ - s имеет значение "size" (размер).

-t

Тег (tag) (с аргументом). Назвать расположение или передать программе строку, используемую в качестве ключа поиска. Главным образом используется с текстовыми редакторами и программами просмотра. Примеры:

cvs(1), ex(1), less(1), vi(1) .

-u

Пользователь (user) (с аргументом). Указать пользователя по имени или числовому идентификатору (UID). Примеры:

crontab(1), emacs(1), fetchmail(1), fuser(1), ps(1) .

-v

Вывод подробных сведений (verbose) (с аргументом или без). Используется для включения мониторинга транзакций, более объемных списков или вывода отладочных сообщений. Примеры:

cat(1), cp(1), flex(1), tar(1) и многие другие.

Версия (version) (без аргумента). Отображает версию программы на стандартном выводе и завершает свою работу. Примеры:

cvs(1), chattr(1), patch(1), uucp(1) . В более распространенном варианте данное действие вызывается ключом -V.

Версия (version) (без аргумента). Отображает версию программы на стандартный вывод и завершает работу (кроме того, часто распечатывает на экране детали конфигурации, с которой была скомпилирована программа). Примеры:

gcc(1), flex(1), hostname(1) и многие другие. Было бы крайне неожиданно, если бы данный ключ использовался иным способом.

-w

Ширина (width) (с аргументом). Главным образом используется для определения ширины в форматах вывода. Примеры:

faces(1), grops(1), od(1), pr(1), shar(1) .

Предупреждение (warning) (без аргументов). Включает диагностические предупреждения или подавляет их. Примеры:

fetchmail(1), flex(1), nsgmls(1) .

-x

Разрешить отладку (с аргументом или без). Данный ключ подобен -d. Примеры:

sh(1), uucp(1) .

Извлечь (extract) (с аргументом). В качестве аргумента указываются файлы, которые необходимо извлечь из архива или рабочего комплекта. Примеры:

tar(1), zip(1) .

-y

Да (yes) (без аргументов). Санкционирует потенциально разрушительные действия, для которых программа обычно запрашивает подтверждение. Примеры:

fsck(1), rz(1) .

-z

Разрешает сжатие (без аргументов). Часто данный ключ используется в программах архивирования и резервного копирования. Примеры:

bzip(1) , GNU

tar(1), zcat(1), zip(1), cvs(1) .

Приведенные выше примеры взяты из инструментария Linux, но соответствуют большинству современных Unix-систем.

Выбирая буквы для параметров командной строки разрабатываемой программы, следует обратиться к справочным руководствам по подобным инструментам и попытаться использовать такие же буквы, какие используются в них для аналогичных функций разрабатываемой программы. Следует заметить, что некоторые отдельные прикладные области имеют особенно жесткие соглашения о ключах командной строки, нарушать которые опасно. Такие соглашения характерны для компиляторов, почтовых агентов, текстовых фильтров, сетевых утилит и программного обеспечения для X Window. Например, разработчик, который написал почтовый агент, использующий ключ -s для какой-либо другой функции вместо подстановки темы, услышит справедливую критику в свой адрес.

Стандарты программирования проекта GNU[100] рекомендуют традиционные значения для нескольких параметров с двойным дефисом. В проекте также перечислены длинные параметры, которые используются во многих GNU-программах, хотя они и не стандартизированы. Если программа разрабатывается с использованием GNU-стиля, и какой-либо необходимый параметр обладает функцией, подобной одному из перечисленных параметров, то следует всеми способами соблюсти правило наименьшей неожиданности и использовать имеющееся имя.

10.5.2. Переносимость на другие операционные системы

Для применения параметров командной строки необходимо ее присутствие в системе. В семействе MS-DOS командная строка, несомненно, присутствует, хотя в Windows она скрыта GUI-интерфейсом и используется слабо. Тот факт, что символом параметра обычно является

"/" вместо "-" — только деталь. Классическая MacOS и другие исключительно GUI-среды не имеют близкого эквивалента параметров командной строки.

10.6. Выбор метода

Выше последовательно были рассмотрены системные и пользовательские файлы конфигурации, переменные окружения и аргументы командной строки — направление от конфигурации, которую изменить сложнее, к конфигурации, изменяемой проще. Имеется четкое соглашение о том, что удобные Unix-программы, использующие несколько мест хранения конфигурационных сведений, должны просматривать их в указанном порядке, позволяя более поздним установкам переназначать более ранние (существуют специфические исключения, такие как параметры командной строки, определяющие местонахождение файла профиля).

В частности, настройки окружения обычно имеют превосходство над установками файлов конфигурации, но могут быть переназначены параметрами командной строки. Хорошей практикой является предоставление параметра командной строки, подобного ключу -e в утилите

make(1) , который может переназначить установки окружения или объявления в конфигурационных файлах. Данный способ позволяет включать программу в сценарии со строго определенным поведением независимо от вида файлов конфигурации или значений переменных окружения.

Выбор мест хранения настроек зависит от количества постоянных конфигурационных параметров, которые должны сохраняться между вызовами программы. Программы, разработанные преимущественно для использования в неинтерактивном режиме (как, например, генераторы или фильтры в конвейерах), обычно полностью конфигурируются с помощью параметров командной строки. В число хороших примеров данной модели включаются утилиты

ls(1), grep(1) и

sort(1) . Другой крайний случай — крупные программы со сложным диалоговым поведением могут полностью зависеть от файлов конфигурации и переменных окружения, и в обычном использовании требуют только несколько параметров командной строки или не требуют их вообще. Большинство диспетчеров окон системы X представляют собой хорошие примеры такой модели.

(В операционной системе Unix файл может иметь несколько имен или "ссылок". Во время запуска каждая программа может определить имя, посредством которого она была вызвана. Другой способ указать программе, имеющей несколько режимов работы, в каком из них она должна запуститься, заключается в создании ссылки для каждого режима. Программа обнаруживает ссылку, через которую она была вызвана, и соответствующим образом изменяет режим работы. Однако данная техника обычно считается неаккуратной и используется нечасто.)

Ниже рассматриваются две программы, которые учитывают информацию, собранную из всех трех групп конфигурационных данных. Читателям будет полезно обдумать, почему для каждого блока конфигурационных данных информация накапливается именно таким способом.

10.6.1. Учебный пример:

fetchmail

Программа

fetchmail использует только две переменные среды — USER и HOME. Данные переменные входят в состав предопределенного набора конфигурационных данных, инициализируемых системой, и используются многими программами.

Значение переменной HOME используется для поиска файла профиля .fetchmailrc, который содержит конфигурационную информацию, описанную с помощью довольно сложного синтаксиса, подчиняющегося shell-подобным лексическим правилам, описанным выше. В данном случае такой подход целесообразен, поскольку после первоначальной установки конфигурация fetchmail изменяется исключительно редко.

Не существует ни /etc/fetchmailrc, ни какого-либо другого специфичного для fetchmail общесистемного файла. Обычно в таких файлах содержится конфигурация, которая не является специфичной для отдельного пользователя. В fetchmail действительно используется небольшая группа подобных свойств, в частности, имя локального почтмейстера (postmaster), несколько ключей и значений, описывающих настройку локального почтового транспорта (например, номер порта локального SMTP-слушателя). Однако на практике данные значения редко отклоняются от стандартных значений, настроенных на этапе компиляции. Если они изменяются, то это часто происходит определенным для пользователя способом. Поэтому в общесистемном конфигурационном файле fetchmail нет никакой необходимости.

Программа fetchmail может извлекать тройки узел/ имя/пароль из файла .netrc. Таким образом, программа получает сведения аутентификации наименее неожиданным способом.

В fetchmail имеется развитый набор параметров командной строки, близко, но не полностью дублирующих установки, которые могут быть выражены в файле .fetchmailrc. Данный набор первоначально не был большим, однако со временем он разрастался, по мере того как в мини-язык .fetchmailrc добавлялись новые конструкции, а также более или менее автоматически добавлялись параллельные параметры командной строки.

Цель поддержки всех данных параметров заключалась в том, чтобы сделать программу fetchmail более простой для включения в сценарии, позволяя пользователям переназначать с помощью командной строки настройки, определенные в файлах конфигурации. Но оказалось, что за исключением нескольких параметров, таких как --fetchall и -verbose, требуются очень не многие имеющиеся параметры. Кроме того, в ходе использования программы вообще не возникает потребностей, которые невозможно было бы удовлетворить с помощью сценария, создающего на лету временный конфигурационный файл и передающего его в fetchmail с помощью ключа -f.

Таким образом, большинство параметров командной строки никогда не используются, и их включение, вероятно, было ошибкой. Они несколько загромождают код fetchmail, не давая какого-либо полезного эффекта.

Если бы загромождение кода было единственной проблемой, то никто, кроме нескольких кураторов, не беспокоился бы. Однако параметры увеличивают вероятность появления ошибок в коде, особенно ввиду непредвиденного взаимодействия между редко используемыми параметрами. Еще хуже то, что они значительно загромождают справочные

руководства, которые обременяют всех. Дуг Макилрой

Из всего вышесказанного можно извлечь следующий урок: если бы я достаточно тщательно продумал модель использования программы fetchmail и меньше придерживался специализированного подхода в добавлении функций, то дополнительной сложности можно было бы избежать.

Альтернативный способ исправления подобных ситуаций, который не загромождает ни код, ни руководство, заключается в использовании ключа "установить переменную параметра", такого как ключ -O в

sendmail, позволяющий указать имя и значение параметра и устанавливающий значение данного параметра так, как если бы оно было задано в конфигурационном файле. Более мощный вариант данной функции имеется в программе

ssh с (ключ -o): аргумент ключа -o интерпретируется так, как будто он представляет собой строку, добавленную в конец конфигурационного файла. При создании аргумента можно использовать весь доступный конфигурационный синтаксис. Любой из таких подходов предоставляет пользователям с необычными требованиями способ переназначать конфигурацию из командной строки, не требуя при этом от разработчика создания отдельного параметра для каждого конфигурационного значения.Генри Спенсер.

10.6.2. Учебный пример: сервер XFree86

Система оконного интерфейса X представляет собой ядро, поддерживающее растровые дисплеи на Unix-машинах. Unix-приложения, запущенные на клиентской машине с растровым дисплеем, получают входные события посредством системы X и отправляют ей запросы на создание экранных изображений. Многих ставит в тупик то, что "серверы" X фактически запускаются на клиентской машине — они существуют для обслуживания запросов на взаимодействие с дисплеем клиентской машины. Приложения, отправляющие такие запросы X-серверу, называются "X-клиентами", несмотря на то, что они могут быть запущены на серверной машине. Невозможно без путаницы объяснить эту "перевернутую" терминологию.

X-серверы имеют неприступно сложный интерфейс к своему окружению. Это не удивительно, поскольку им приходится взаимодействовать с широким диапазоном сложного аппаратного обеспечения и пользовательских настроек. Запросы окружения являются общими для всех X-серверов, документированными в справочных руководствах

X(1) и

Xserver(1), а значит, представляют собой полезный пример для изучения. Ниже рассматривается XFree86, реализация системы X, применяемая в операционной системе Linux и других Unix-системах с открытым исходным кодом.

Во время запуска сервер XFree86 изучает общесистемный конфигурационный файл. Точный путь к данному файлу варьируется в различных сборках X на разных платформах, но базовое имя постоянно — XF86Config. Данный файл имеет shell-подобный синтаксис, как было описано выше. Ниже приводится пример одного из разделов файла XF86Config. Пример 10.2. Конфигурация X

```
# VGA-сервер 16 цветов

Section "Screen"
```

```
 Driver "vga16"

 Device "Generic VGA"

 Monitor "LCD Panel 1024x768"

 Subsection "Display"

  Modes "640x480" "800x600"

  Viewport 0 0

 EndSubsection

EndSection
```

В файле XF86Config описывается аппаратное обеспечение машины (графическая плата, монитор), клавиатура и указательное устройство (мышь/трекбол/сенсорная панель). Хранение данных сведений в общесистемном конфигурационном файле целесообразно, поскольку они касаются всех пользователей машины.

Как только X-сервер получил конфигурацию аппаратного обеспечения из конфигурационного файла, он использует значение переменной окружения HOME для обнаружения двух файлов профиля, расположенных в начальном каталоге вызывающего пользователя, .Xdefaults и .xinitrc[101].

В файле .Xdefaults указываются специализированные ресурсы данного пользователя, относящиеся к системе X (в число простейших примеров можно включить шрифт и цвета фона и переднего плана для эмулятора терминала). Однако выражение "относящиеся к системе X" указывает на проблему проектирования. Группировать описания всех данных ресурсов в одном месте удобно для проверки и редактирования, но не всегда ясно, какие ресурсы следует объявлять в файле .Xdefaults, и какие параметры необходимо хранить в файле профиля приложения. В файле .xinitrc определяются команды, которые должны быть выполнены для инициализации пользовательского рабочего стола в X сразу после запуска сервера. В число данных программ почти всегда включается диспетчер окон или диспетчер сеанса.

X-серверы имеют большой набор параметров командной строки. Некоторые из них, такие как -fp (font path — путь к шрифтам), имеют преимущество перед параметрами файла XF86Config. Некоторые, например, -audit, предназначены для того, чтобы облегчить отслеживание ошибок сервера. Если данные параметры вообще используются, то, скорее всего, очень часто изменяются между тестовыми запусками, а следовательно, являются маловероятными кандидатами на включение в конфигурационный файл. Очень важным является параметр, устанавливающий номер дисплея сервера. На узле могут быть запущены несколько серверов, если каждому из них предоставляется уникальный номер дисплея, но все экземпляры совместно используют один и тот же конфигурационный файл (или файлы), поэтому номер дисплея невозможно получить исключительно из данных файлов.

10.7. Нарушение правил

Описанные в данной главе соглашения не абсолютны, но их нарушение в будущем усугубить разногласия между пользователями и разработчиками. В случае крайней необходимости их можно нарушить, однако, прежде чем это сделать, необходимо точно знать, для чего это

необходимо. Разработчику, нарушающему данные соглашения, следует убедиться, что решение задач традиционными способами невозможно и что он имеет полное представление об ошибках в соответствии с правилом исправности.

11

Интерфейсы: модели проектирования пользовательских интерфейсов в среде Unix

Начало всех наших знаний в наших ощущениях. —Леонардо да Винчи

Интерфейсом программы является совокупность всех способов, посредством которых программа обменивается данными с пользователями и другими программами. В главе 10 рассматривалось применение переменных окружения, ключей, файлов конфигурации и других элементов интерфейсов запуска. В данной главе эта тема развивается далее и поясняется работа Unix-интерфейсов после запуска. Поскольку создание кода пользовательского интерфейса обычно занимает не менее 40% времени разработки, знание хороших конструкторских моделей в этой области особенно важно, так как позволяет избежать многих неудачных попыток и переделок.

Для Unix-традиции проектирования интерфейсов всегда были характерны две идеи. Одна из них состоит в заблаговременном проектировании обмена данными с другими программами, а другая — это правило наименьшей неожиданности.

Использование синергетических комбинаций Unix-программ может предоставить дополнительный выигрыш. Различные методы создания таких комбинаций рассматривались в главе 7. "Другие программы" в проектировании интерфейсов в Unix — не запоздалое дополнение и не предельный случай, как во многих других операционных системах. Данная часть проектирования интерфейсов скорее является центральной проблемой, которую необходимо сбалансировать и взвешенно интегрировать с пользовательскими требованиями к конструкции интерфейсов.

Многое в традиции Unix-сообщества, касающейся проектирования интерфейсов программ, при первом рассмотрении может показаться чуждым и бессистемным или даже (в контексте эпохи GUI-интерфейсов) полностью регрессивным. Однако несмотря на различные недостатки и неупорядоченность, данная традиция обладает внутренней логикой, которая достойна изучения и понимания. Она отражает эмпирические правила, накопленные за годы долгой истории Unix, касающиеся способов обеспечения эффективного взаимодействия, как с пользователями, так и с другими программами. Традиция включает в себя множество соглашений, которые позволяют унифицировать программы — она определяет "наименее неожиданные" альтернативы для широкого диапазона общих проблем проектирования интерфейсов.

После запуска программа обычно получает входные данные или команды из следующих источников:

• данные и команды, представленные на стандартном вводе программы;

• входные данные, переданные посредством IPC-методов, такие как события X-сервера и сетевые сообщения;

• файлы и устройства с известным расположением (например, имя файла данных,

переданное или вычисленное программой).

Теми же путями программы могут отправлять данные (вывод направляется на стандартный вывод).

Некоторые Unix-программы являются графическими, некоторые имеют символьные экранные интерфейсы, а в некоторых используется простейшая конструкция текстового фильтра, которая не изменилась со времен механических телетайпов. Непосвященным далеко не очевидно, почему в какой-либо программе используется тот или иной стиль, или, вернее, почему в Unix вообще поддерживается такое многообразие стилей интерфейса.

В Unix существует несколько конкурирующих стилей интерфейса. Все они до сих пор существуют по определенным причинам. Они оптимизированы для использования в различных ситуациях. Поняв соответствие между задачей и стилем интерфейса, можно научиться выбирать наиболее подходящий стиль для выполнения необходимой работы.

11.1. Применение правила наименьшей неожиданности

Правило наименьшей неожиданности представляет собой общий принцип проектирования всех видов интерфейсов, а не только программных: "делайте наименее неожиданные вещи". Данное правило — следствие того факта, что человек способен уделять внимание одновременно только одному объекту (см. книгу

"The Humane Interface" [64]). Неожиданности в интерфейсе притягивают внимание к самому интерфейсу, а не к задаче, для решения которой он предназначен.

Таким образом, наилучший способ разработки удобных интерфейсов заключается в том, чтобы по возможности никогда не разрабатывать полностью новую модель интерфейса. Нововведения — входные препятствия для пользователя. Они повышают нагрузку на пользователя, поскольку требуют изучения, поэтому их количество должно быть минимальным. Вместо этого, следует взвешенно оценить опыт и знания пользовательского контингента и попытаться найти функциональное сходство между разрабатываемой программой и программами, которые пользователи, вероятнее всего, уже изучили, а затем сымитировать соответствующие части существующих интерфейсов.

Правило наименьшей неожиданности не следует трактовать как призыв к механическому консерватизму в проектировании. Нововведения повышают издержки нескольких первых попыток взаимодействия пользователя с интерфейсом, но неразвитая конструкция навсегда сделает интерфейс излишне трудным. Как и в других видах проектирования, правила не заменят хорошего вкуса и инженерного подхода. Необходимо тщательно взвешивать компромиссы и рассматривать их с точки зрения

пользователя . Осознанный уклон в сторону правила наименьшей неожиданности — хорошая практика, и главным образом потому, что разработчики интерфейсов (как и другие программисты) имеют несознательную тенденцию быть "излишне умными" для блага пользователя.

Одно из следствий правила наименьшей неожиданности: при любой возможности позволять пользователю делегировать функции интерфейса хорошо знакомой программе. В главе 7 уже отмечалось, что если разрабатываемая программа требует от пользователя редактирования значительного количества текста, то ее следует писать так, чтобы она вызывала редактор (указанный пользователем), а не создавать собственный интегрированный редактор. Такой

подход дает возможность

пользователям , которым их предпочтения известны лучше, чем разработчикам, выбрать наименее неожиданную альтернативу.

В данной книге уже пропагандировался симбиоз и делегирование как тактики, стимулирующие повторное использование кода и сокращение сложности. Суть данных методик заключается в том, что если пользователи могут перехватить делегирование и направить его избранному агенту, то это не просто экономически выгодно для разработчика, но и активно расширяет возможности пользователей.

В случае, когда делегирование невозможно, следует воспроизводить удачные решения. Целью правила наименьшей неожиданности является сокращение общей сложности, которую необходимо постичь пользователю, для того чтобы использовать интерфейс. Относительно примера с текстовым редактором, это означает, что если реализовать встроенный редактор действительно необходимо, то будет лучше, если команды этого редактора будут представлять собой подмножество команд широко известного универсального редактора (или нескольких редакторов; в оболочках bash и ksh имеются CLI-редакторы, которые позволяют пользователям выбирать стиль редактирования

vi или

Emacs .)

Например, в Unix-версиях Web-браузеров Netscape и Mozilla текстовые поля форм распознают подмножество стандартных клавиатурных комбинаций для редактора

Emacs . Control-A переводит курсор в начало строки, Control-D удаляет следующий символ и так далее. Такой вариант помогает пользователям, знакомым с

Emacs , а остальным позволяет использовать не худший, уникальный набор команд. Единственным способом улучшить его был выбор клавиатурных комбинаций, связанных с каким-либо редактором, получившим более широкое распространение, чем

Emacs . Но среди первых пользователей Netscape такой редактор не использовался.

Данные принципы применимы во многих других областях проектирования интерфейсов. Они, в частности, означают, что было бы крайне недальновидно создавать для интерактивной справочной системы нестандартные форматы документов, тогда как пользователям удобно использовать Web-браузер, распознающий HTML. Даже при разработке аркадной игры целесообразно рассмотреть интерфейсы предыдущих игр, для того чтобы понять, можно ли обеспечить новым пользователям чувство комфорта, позволяя им перенести накопленные в других играх навыки управления.

11.2. История проектирования интерфейсов в Unix

Операционная система Unix предшествует современному стилю проектирования программных интерфейсов с интенсивным применением графики. Более 10 лет после появления первой Unix-системы в 1969 году на телетайпах и "немых" текстовых терминалах нормой был интерфейс командной строки (Command-Line Interface — CLI). Большая часть базового инструментария Unix (такие программы как

ls(1), cat(1) и

grep(1)) до сих пор отражает это наследие.

После 1980 года в Unix постепенно развивалась поддержка прорисовки экранов на символьных терминалах. В программах начали смешивать интерфейс командной строки с визуальным интерфейсом. Часто общим командам присваивались клавиатурные комбинации, которые не отражались на экране. Некоторые ранние программы, написанные в этом стиле (они часто называются curses-программами по названию библиотеки управления курсором для прорисовки экрана, которая обычно применялась в их реализации, или rogue-подобными по названию первого приложения, использовавшего библиотеку curses), используются до сих пор. В число известнейших примеров входят ролевая игра

rogue(1) , текстовый редактор

vi(1) , а также почтовая программа

elm(1) (вышедшая несколькими годами позднее) и ее современный потомок

mutt(1) .

Через несколько лет, в середине 80-х годов прошлого века, весь компьютерный мир начал усваивать результаты передовых разработок графических пользовательских интерфейсов (Graphical User Interface — GUI), которые с начала 70-х годов продолжались в исследовательском центре компании Xerox Palo Alto Research Center (PARC). В области персональных компьютеров работа Xerox PARC вдохновила создание интерфейса Apple Macintosh, а через него и дизайна Microsoft Windows. Адаптация данных идей в операционной системе Unix пошла значительно более сложным путем.

Приблизительно в 1987 году система X Window обошла несколько более ранних конкурирующих и опытных проектов и стала стандартным средством графического интерфейса для операционной системы Unix. Споры о том, хорошо это было или плохо, продолжаются до сих пор. Возможно, некоторые из конкурентов X Window (особенно Network Window System или NeWS разработки Sun) были гораздо более мощными и изящными. Система X, однако, имела одно неоспоримое преимущество — открытый исходный код. Ее код разрабатывался в Массачусетском технологическом университете исследовательской группой, более заинтересованной изучением проблемной области, а не созданием продукта, и остался свободно распространяемым и модифицируемым. В силу данных причин код X оказался способным

привлечь поддержку со стороны широкого круга разработчиков и спонсирующих корпораций, которые не хотели оставаться позади закрытого продукта одного поставщика. (Это, несомненно, послужило прообразом важной идеи в прорыве операционной системы Linux десятью годами позже.)

Разработчики X в самом начале решили, что система будет поддерживать "механизм, а не политику". Их целью было сделать систему X настолько гибкой и переносимой между платформами, насколько это возможно, одновременно внося как можно меньше ограничений на вид и восприятие X-программ. Было решено, что вид и восприятие интерфейса будут поддерживаться "инструментариями" — библиотеками, вызывающими X-службы, которые связаны с пользовательскими программами. Система X должна была быть спроектирована для поддержки множества диспетчеров окон[102] и не требовала бы от диспетчера окон наличия каких-либо особых привилегий или уникально тесной интеграции с механизмом системы X.

Данный подход был диаметрально противоположным принятому в коммерческих продуктах Macintosh и Windows, которые навязывали определенный вид и восприятие интерфейса, непосредственно встраивая его в систему. Различие в подходах обеспечило системе X

долгосрочные эволюционные преимущества — она оставалась легко адаптируемой по мере открытия новых нюансов человеческого фактора в конструкции интерфейсов. Однако различие подходов также обеспечило последующее разделение мира X вследствие разнообразия инструментариев оконных менеджеров, а также из-за многих экспериментов с внешним видом и восприятием интерфейса.

С середины 90-х годов прошлого века система X стала использоваться повсеместно даже на низкопроизводительных персональных Unix-машинах. Использование операционной системы Unix с текстовых терминалов, в противоположность графическим компьютерным консолям, резко пошло на спад и, вероятно, достигло своего заката. Соответственно, использование псевдографических интерфейсов для новых приложений также пошло на убыль. Большинство новых приложений, которые прежде разрабатывались бы в данном стиле, в настоящее время используют X-инструментарий. Полезно отметить, что более старая традиция Unix, традиция использования CLI-дизайна, до сих пор весьма сильна и во многих областях и успешно конкурирует с X.

Также полезно отметить факт существования нескольких специфических прикладных областей, в которых псевдографические интерфейсы curses- или rogue-подобного стиля остаются нормой — особенно текстовые редакторы и интерактивные коммуникационные программы, такие как почтовые агенты, программы чтения новостей и chat-клиенты.

Таким образом, по историческим причинам в Unix-программах имеется широкий диапазон стилей интерфейсов. Строчные, псевдографические экранные и интерфейсы на основе системы X. Мир X-интерфейсов отчасти разделен конкуренцией между различными видами X-инструментария и диспетчерами окон (хотя в настоящее время данная проблема стоит уже не так остро, как несколько лет назад).

11.3. Оценка конструкций интерфейсов

Все данные стили интерфейсов продолжают существовать до сих пор благодаря тому, что они адаптированы для различных задач. Принимая решение о конструкции проекта, важно знать, как подбирать (или комбинировать) приемлемые стили для данного приложения и его пользователей.

Ниже для разделения стилей интерфейсов на категории используются пять базовых показателей:

лаконичность, выразительность, простота, прозрачность и

возможность использования в сценариях . Некоторые из данных понятий уже упоминались ранее в данной книге, здесь даны их определения. Данные показатели являются сравнительными, а не абсолютными, и оценивать их необходимо в контексте определенной проблемной области, имея некоторые сведения об уровне навыков пользователей. Тем не менее, данные показатели способствуют в формировании конструктивного мышления.

Интерфейс программы является "лаконичным", если максимальная длительность и сложность действий, которые необходимо выполнить для совершения транзакции с данным интерфейсом, не высоки (мерой может послужить количество нажатий клавиш, жестов или секунд необходимого внимания). В лаконичных интерфейсах множество усилий упаковано в относительно небольшом числе изменений состояния.

Интерфейсы являются "выразительными", когда их можно без трудностей использовать для

управления целым рядом действий.

Наиболее выразительные интерфейсы могут управлять комбинациями действий, которые не предусмотрены разработчиком программы, но, тем не менее, предоставляют пользователю полезные и последовательные результаты.

Различие между лаконичностью и выразительностью представляется весьма важным. Рассмотрим два различных способа ввода текста: с клавиатуры или путем выбора с помощью мыши символов на экране дисплея. Оба способа имеют равную выразительность, однако клавиатура предоставляет более лаконичный способ (это можно легко проверить, сравнив средние значения скорости ввода текста). С другой стороны, рассмотрим два диалекта одного языка программирования, один с типом комплексных чисел, а другой без него. Внутри общей проблемной области их лаконичность идентична, но для математиков или инженеров электротехников диалект с комплексными числами будет более выразительным.

"Простота" интерфейса обратно пропорциональна мнемонической нагрузке на пользователя — количество элементов (команд, жестов, основных понятий), которые должен запомнить пользователь специально для работы с данным интерфейсом. Программные языки характеризуются высокой мнемонической нагрузкой и низкой простотой, меню и экранные кнопки с понятными надписями более просты.

Идее прозрачности ранее была отведена целая глава, в которой она рассматривалась через призму прозрачности интерфейса, и как замечательный пример прозрачности описывался

редактор аудиофайлов audacity . Однако тогда больше внимания было уделено прозрачности другого вида, относящейся скорее к структуре кода, а не к структуре пользовательского интерфейса. Таким образом, прозрачность пользовательского интерфейса описывалась скорее в аспекте его эффективности (ничто не вторгается между пользователем и проблемной областью), чем в специфических особенностях создающей его конструкции. В данной главе основное внимание уделяется именно специфическим особенностям.

Понятие прозрачности интерфейса — определяется тем, как мало сведений об условиях проблемы, данных или программы необходимо помнить пользователю во время

использования интерфейса. Интерфейс обладает высокой прозрачностью, когда он естественно представляет промежуточные результаты, полезную обратную связь и уведомления об ошибках в результате действий пользователя. Так называемые интерфейсы WYSIWYG (What You See Is What You Get — что видишь, то получаешь) предназначены для создания максимальной прозрачности, однако иногда наблюдается обратный эффект, в частности создание чрезмерно упрощенного вида проблемной области.

К конструированию интерфейсов также применимо родственное понятие воспринимаемости. Воспринимаемый интерфейс предоставляет пользователю помощь в его изучении, например, приветственное сообщение, указывающее на контекстную справку, или всплывающие надписи с содержательными пояснениями. Несмотря на то, что воспринимаемость должна реализовываться довольно разными способами для каждого стиля интерфейса, которые будут рассматриваться далее, степень ее досягаемости почти совсем не зависит от стиля интерфейса. Поэтому как показатель, воспринимаемость в данном обсуждении не используется.

Необходимо отметить, что прозрачность кода и конструкции автоматически не означает прозрачность интерфейса и наоборот. Достаточно просто найти код, который имеет одно качество, но не имеет другого.

"Возможность использования интерфейса в сценариях" — легкость, с которой другие программы могут манипулировать данным интерфейсом (например, посредством

IPC-механизмов, рассмотренных в главе 7). Программы, имеющие такую возможность, могут легко использоваться другими программами как компоненты, что сокращает необходимость дорогостоящего специального программирования и относительно упрощает автоматизацию повторяющихся задач.

Последний момент — автоматизация повторяющихся задач — заслуживает большего внимания, чем ему обычно уделяется. У Unix-программистов, администраторов и пользователей развивается привычка продумывать используемые рутинные процедуры, а затем организовывать их так, что им более не приходится выполнять данные действия вручную или даже думать о них. Такая привычка зависит от интерфейсов, которые можно использовать в сценариях. Этот незаметный, но чрезвычайно мощный усилитель продуктивности не доступен в большинстве других программных сред.

Полезно учитывать то, что работа компьютерных программ и работа пользователей зависит от описываемых показателей абсолютно по-разному. То же можно сказать о новичках и опытных пользователях в отдельной проблемной области. Ниже рассматривается, как изменяются компромиссы между этими показателями для различных контингентов пользователей.

11.4. Компромиссы между CLI- и визуальными интерфейсами

После ухода телетайпов CLI-стиль ранней Unix в течение долгого времени остается полезным по двум причинам. Первая заключается в том, что интерфейсы командной строки и командных языков являются более выразительными, чем визуальные, особенно для сложных задач. Другая причина состоит в том, что CLI-интерфейсы очень хорошо используются в сценариях — они легко поддерживают комбинирование программ, как подробно разъяснялось в главе 7. Обычно (хотя и не всегда) CLI-интерфейсы также более лаконичны.

Недостатком CLI-стиля, несомненно, является то, что он почти всегда представляет высокую мнемоническую нагрузку (низкую простоту) и обычно имеет слабую прозрачность. Большинство пользователей (особенно нетехнические конечные пользователи) находят такие интерфейсы довольно непонятными и сложными в изучении.

С другой стороны, "дружественные" GUI-интерфейсы других операционных систем имеют свои собственные проблемы. Поиски необходимых кнопок подобны игре в "Adventure": интерфейсы в этих системах настолько же трудны, насколько любой интерфейс командной строки в Unix, за исключением того, что пользователь теоретически может "найти сокровища" после достаточного исследования. В Unix пользователю необходимо руководство. Брайан Керниган.

Запросы к базам данных представляют собой хороший пример интерфейса, для которого нажатие клавиш является не просто обременительным, а крайне ограничивающим. Ни клавиатурные комбинации в полноэкранном символьном интерфейсе, ни GUI-жесты на графическом дисплее не способны выразить обычные действия в данной предметной области так же выразительно или лаконично, как передача SQL-запроса непосредственно серверу. И, безусловно, проще заставить клиентскую программу генерировать SQL-запросы, чем заставить ту же программу имитировать действия пользователя в GUI-интерфейсе.

С другой стороны, многие нетехнические пользователи баз данных настолько не приемлют необходимость запоминания SQL-синтаксиса, что предпочитают менее лаконичный и менее выразительный полноэкранный или GUI-интерфейс.

SQL является хорошим примером, иллюстрирующим другой момент. Большинство мощных CLI-интерфейсов представляют собой не уникальные наборы команд, а императивные мини-языки, разработанные согласно описанным в главе 8 принципам. Данные мини-языки находятся на самой мощной и самой сложной границе спектра CLI-интерфейсов. Они достигают максимальной выразительности, но сводят к минимуму простоту. Их трудно использовать, и обычно необходимо осмотрительно скрывать их от обычных конечных пользователей, однако они представляют собой идеальное решение там, где мощность и гибкость интерфейса является наиболее важной. Если они разработаны соответствующим образом, то их также намного проще использовать в сценариях.

Некоторые приложения, в отличие от запросов к базам данных, являются естественно визуальными. К ним относятся программы для рисования, Web-браузеры и презентационное программное обеспечение. Данные прикладные области имеют два общих аспекта — (a) прозрачность в них крайне важна и (b) простейшие действия в предметной области сами по себе являются визуальными: "нарисовать", "показать выбранный объект", "переместить объект".

Оборотной стороной программ для рисования является то, что в них трудно фиксировать связи внутри изображений, которыми они манипулируют. Требуется тщательное, продуманное проектирование, для того чтобы, например, предоставить пользователю какой-либо указатель на структуру изображений с повторяющимися элементами. Это является общей проблемой проектирования визуальных интерфейсов.

В главе 6 рассматривался редактор звуковых файлов Audacity. Конструкция его интерфейса имеет успех, поскольку она выполняет определенно четкую задачу преобразования предметной области, т.е. звука в простой набор визуальных представлений (заимствованных с индикаторов эквалайзеров на стереосистемах). Это достигается полным воспроизведением результатов преобразования звуков в изображения колебаний. Визуальные операции не просто подогнаны под трансформацию звука. Все они связаны с данным преобразованием.

Однако в приложениях, которые

не являются естественно визуальными, визуальные интерфейсы наиболее целесообразны для простых единичных или нечастых заданий, выполняемых начинающими пользователями (что иллюстрируется примером баз данных).

Сопротивление CLI-интерфейсам часто уменьшается по мере того, как пользователи становятся более опытными. Во многих предметных областях пользователи (особенно те, которые используют программы

регулярно) достигают того момента, когда лаконичность и выразительность CLI становится более ценной, чем избежание мнемонической нагрузки. Поэтому, например, начинающие пользователи компьютеров предпочитают рабочие столы GUI, а опытные пользователи часто постепенно обнаруживают, что они предпочитают ввод команд в оболочке.

CLI-интерфейсы также часто становятся полезными, по мере того как проблемы расширяются и вовлекают больше заранее подготовленных, процедурных и повторяющихся действий. Поэтому, например, настольная издательская программа WYSIWYG-типа обычно предоставляет простейший способ создания сравнительно небольших и неструктурированных документов, таких как деловые письма. Однако для сложных документов, сравнимых по размеру с книгами, которые составляются из разделов и во время компоновки могут потребовать множество глобальных изменений формата или структурных манипуляций, такой форматер, как

troff, Tex или какой-либо процессор XML-разметки, является обычно более эффективным вариантом (более подробное обсуждение данного компромисса приведено в главе 18).

Даже в тех сферах, которые являются естественно визуальными, расширение проблемы часто склоняет выбор компромиссного решения в пользу CLI-интерфейса. Если требуется загрузить и сохранить одну Web-страницу с заданного URL, то выбор и щелчок мыши (или ввод и щелчок) будет превосходным способом. Однако для Web-форм потребуется использование клавиатуры. И если необходимо загрузить и сохранить страницы, соответствующие заданному списку из 50 URL-адресов, то CLI-клиент, способный считывать URL со стандартного ввода или из командной строки, позволит сократить количество излишних движений.

В качестве другого примера рассмотрим изменение таблицы цветов в графическом изображении. Если требуется изменить один цвет (например, осветлить его на величину, которую можно определить только посмотрев на изображение), то визуальное диалоговое окно со средствами подбора цветов является почти обязательным. Однако предположим, что необходимо заменить всю таблицу набором указанных RGB-значений или создать и проиндексировать большое количество пиктограмм. Это те операции, в которых GUI-интерфейсам обычно не хватает выразительности. Даже если такая выразительность достигнута, вызов соответствующим образом разработанного CLI-интерфейса или программы фильтра позволит выполнить задачу гораздо более лаконичным способом.

Наконец, (как отмечалось выше) CLI-интерфейсы являются важным средством, облегчающим использование одних программ из других. Графический редактор с GUI-интерфейсом, который

способен создавать пакеты пиктограмм для списка файлов, вероятно, выполнит данную задачу с помощью дополнительного приложения, написанного на языке сценариев, вызывающего внутренний CLI-интерфейс графического редактора (как, например, средство script-fu для GIMP). Unix-среды настолько повышают ценность CLI-интерфейсов именно потому, что их IPC-средства хорошо развиты, имеют низкие издержки и легкодоступны из пользовательских программ.

Рост популярности GUI-интерфейсов (начиная с 1984 года) имел неудачное последствие: они затмили достоинства CLI-интерфейсов. Конструкция потребительского программного обеспечения, в частности, стала в большой степени склоняться к GUI-интерфейсам. Несмотря на то, что это удобно для начинающих и случайных пользователей, которые составляют большую часть потребительского рынка, это также несет в себе скрытые затраты для более опытных пользователей, по мере того как они сталкиваются с ограничениями выразительности GUI-интерфейсов. Такие затраты неуклонно увеличиваются по мере того, как пользователи принимаются за решение более сложных проблем. Большая часть затрат связана с тем, что GUI-интерфейсы просто вообще не могут использоваться в сценариях —

каждая операция взаимодействия с ними должна инициироваться пользователем.

Гентнер и Нильсен ярко обобщили данный компромисс в статье

"The Anti-Mac Interface" [28]: "[Визуальные интерфейсы] хорошо подходят для простых действий с небольшим количеством объектов, но по мере того как количество действий или объектов растет, непосредственное управление быстро становится повторяющейся, монотонной работой. Темная сторона интерфейса непосредственного управления заключается в том, что пользователю приходится управлять всем. Вместо руководителя, раздающего высокоуровневые инструкции, пользователь понижается до рабочего конвейера, который должен выполнять одно и то же задание снова и снова". Знаменитый писатель-фантаст Нил Стефенсон (Neal Stephenson) выразил то же мнение не так явно, но более увлекательно в своем ярком и обобщенном очерке

"In the Beginning Was the Command Line" [79].

Подход к данной проблеме типичных представителей старой Unix-школы является значительно менее теоретическим.

Коммерческий мир, как правило, стремится к моде новичков, потому что (a) решения о покупках часто принимаются на основе тридцатисекундного пробного использования и (b) необходимость поддержки пользователей при этом сводится к минимуму, поскольку им предоставляется упрощенный до абсурда GUI-интерфейс. Я нахожу многие не-Unix-системы очень неудобными, поскольку, например, они не предоставляют способа выполнять какие-либо действия с сотней или тысячей файлов; я хочу написать сценарий, но он не поддерживается. Основная проблема таких систем заключается в предположении, что все пользователи постоянно остаются начинающими, и, следовательно, они отвергают Unix, поскольку она в данную модель не вписывается. Майк Леск.

Для того чтобы обеспечить программе долгую жизнь, чтобы программа могла обслуживать как начинающих, так и опытных пользователей, взаимодействовать с другими компьютерными программами, и независимо от того, является ли предметная область естественно визуальной, важна поддержка

обоих видов интерфейсов как CLI-, так и визуальных. История Unix свидетельствует о том, что данная система хорошо подходит для удовлетворения обеих групп потребностей. После представления показательного учебного примера далее в настоящей главе рассматриваются характерные модели проектирования, которые развились в Unix для удовлетворения данных потребностей.

11.4.1. Учебный пример: два способа написания программы калькулятора

Для того чтобы более конкретно изучить преимущества и недостатки GUI- и CLI-интерфейсов, рассмотрим, как данные стили можно полезно применить в конструкции простой интерактивной программы: настольного калькулятора. В качестве примеров для контраста используются утилиты

dc(1)/bc(1) и

xcalc(1) .

Первоначально программой настольного калькулятора в Unix, впервые распространявшегося с Version 7, была утилита

dc(1) — калькулятор с обратной польской нотацией, который мог выполнять арифметические вычисления с неограниченной точностью. Позднее над утилитой

dc(1) был реализован язык алгебраического (с инфиксной нотацией) калькулятора,

bc(1) (взаимосвязь данных программ рассматривалась в качестве учебного примера в главе 7, а затем в главе 8). В обеих программах используется CLI-интерфейс. Пользователь вводит выражение на стандартный ввод и нажимает клавишу Enter, после чего значение выражения печатается на стандартный вывод.

С другой стороны, программа

xcalc(1) визуально имитирует простой калькулятор с кнопками и дисплеем обычного калькулятора.

Подход, примененный в

xcalc(1), проще описать, так как он имитирует интерфейс, с которым знакомы начинающие пользователи. Действительно, в справочном руководстве сказано: "Цифровые клавиши, клавиши +/-, +, -, *, / и = выполняют в точности те функции, которые от них можно ожидать". Все возможности программы передаются надписями на визуальных клавишах. В данном случае соблюдается правило наименьшей неожиданности в строжайшей его форме, а для редких и начинающих пользователей, которым никогда не понадобится читать справочное руководство по использованию данной программы, это представляется реальным преимуществом.

Рис. 11.1. Графический интерфейс программы xcalc

Однако

xcalc также наследует почти полную непрозрачность обычного калькулятора. При расчете сложного выражения невозможно просмотреть и проверить нажатия клавиш, что может оказаться проблемой, если пользователь, например, неверно поставит десятичную точку в выражении, подобном (2.51 + 4.6) * 0.3. В программе не предусмотрено сохранение истории команд, поэтому проверить ввод выражения невозможно. Пользователь получит результат, однако он не будет соответствовать запланированным вычислениям.

Иначе обстоит дело с программами

dc(1) и

bc(1) . Пользователь может исправлять ошибки в выражении по мере его создания. Интерфейс данных программ более прозрачен, поскольку пользователь может просмотреть выполняемые вычисления на каждой стадии. Интерфейс более выразителен, так как интерпретатор

dc/bc , не будучи ограниченным подходящей визуальной моделью калькулятора, может включать в себя гораздо более широкий ассортимент функций (и такие средства, как условные переходы, хранимые переменные и циклы). Следствием данных преимуществ, естественно, является большая мнемоническая нагрузка.

Лаконичность в данном случае — это случайная характеристика. Пользователь, хорошо владеющий навыками машинописи, сочтет CLI-интерфейс более лаконичным, тогда как для не владеющего машинописью пользователя, возможно, будет быстрее выбирать и нажимать клавиши в GUI-интерфейсе. Возможность использования в сценариях совсем не случайна. Комбинацию

dc/bc можно легко использовать в качестве фильтра, а

xcalc вообще невозможно задействовать в сценарии.

Компромисс между простотой для начинающих и полезностью для опытных пользователей в данном случае весьма очевиден. Для нерегулярного использования в ситуациях, когда не трудно проверить ошибку вычисления в уме, более применима программа

xcalc . В более сложных вычислениях, где этапы должны быть не только корректными, но их корректность должна быть

видимой , или в которых значения более удобно вычисляются другой программой, выигрывает комбинация

dc/bc .

11.5. Прозрачность, выразительность и возможность конфигурирования

Unix-программисты унаследовали сильную склонность к созданию выразительных и конфигурируемых интерфейсов. Как и остальные программисты, они думают о том, как привести разрабатываемые интерфейсы в соответствие с требованиями целевой аудитории. Однако они иначе решают вопрос неопределенности целевой аудитории. Разработчики программного обеспечения, главным образом сталкивающиеся с клиентскими операционными системами, обычно склоняются к созданию простых интерфейсов. Они предпочитают жертвовать выразительностью ради простоты. Unix- программисты обычно склонны создавать выразительные и прозрачные интерфейсы и для достижения данных качеств скорее готовы пожертвовать простотой.

Результаты этого часто описываются как интерфейсы, созданные "программистами для программистов". Однако это значительно упрощает проблему. Когда Unix-программист делает выбор в пользу возможности конфигурирования и выразительности вместо простоты, он не обязательно считает свою аудиторию состоящей исключительно из других программистов. Скорее он поступает так инстинктивно, из-за отсутствия знаний о целях конечных пользователей он выбирает наилучший путь — не опекать пользователей и не предугадывать их действия.

Недостатком данной позиции (родственной подходу "механизм, а не политика") является склонность предполагать, что когда высококонфигурируемый и выразительный интерфейс создан, то работа окончена, даже если для кого-либо другого результат почти невозможно использовать без длительного обучения. Обратной стороной конфигурируемости является неотложная потребность в хороших стандартных установках и простом способе устанавливать все настройки в стандартные значения. Обратной стороной выразительности является потребность в руководстве, в самой программе или в документации, где указано, с чего необходимо начинать и как добиваться наиболее распространенных результатов. Генри Спенсер.

Правило прозрачности также оказывает свое влияние. При написании программы, удовлетворяющей требованиям RFC или другого стандарта, который определяет набор управляющих параметров, Unix-программист часто предполагает, что его работа заключается в обеспечении совершенного и прозрачного интерфейса ко всем данным параметрам, причем независимо от того, будет ли, по его мнению, использоваться какой-либо из параметров. Его работа — механизм, а политика принадлежит пользователю.

Эта позиция гораздо меньше допускает неполную поддержку функций. В тех случаях, когда Macintosh или Windows-разработчик сказал бы: "Нам не нужна поддержка данной функции стандарта, большинство пользователей не обратит на это внимания, и она является слишком сложной для них", Unix-разработчик, вероятно, скажет: "Пока не ясно, потребуется ли данная функция или параметр в будущем, следовательно, ее необходимо поддерживать".

Такие противоположные позиции могут приводить к конфликтам, когда Unix-программист работает с не-Unix-программистами, которые, вероятно, истолкуют его выбор конструкции как необдуманное стремление обременять пользователей малопонятными, бессмысленными и даже пугающими техническими подробностями. Mac- и Windows-программисты боятся отпугивать большинство, чтобы служить повышенным потребностям меньшинства.

С другой стороны, Unix-пограммист, вероятно, рассматривает отказ от выразительности как своеобразное бегство от действительности или даже предательство будущих пользователей, которые будут знать о своих требованиях лучше, чем нынешний разработчик. Парадоксально,

но несмотря на то, что позиция Unix-разработчиков часто истолковывается как программистское высокомерие, она в действительности является формой смирения, часто приобретаемой с множеством боевых шрамов.

Противоположные культуры программистов по-разному оценивают степень целесообразности позиции Unix. На какой бы стороне водораздела не оказался читатель, разумным будет умение слушать других и понимание предпосылок, лежащих в основе противоположной точки зрения. Чтобы не попасть в ловушку запугивания пользователей или снисходительного отношения к ним, возможно, следует создавать прозрачные интерфейсы, в которых дополнительные функции присутствуют, но присутствуют незаметно. Хорошими примерами для подражания в данном случае являются программы

audacity и

kmail , рассмотренные в главе 6.

Наконец, приведем замечание относительно конструирования пользовательского интерфейса для нетехнических пользователей. Это сложное искусство, и в нем Unix-программисты не имеют хороших традиций. Однако идеи, которые были сформулированы здесь при изучении Unix-традиций, позволяют сделать одно четкое и полезное утверждение о нем. Когда пользователи говорят, что интерфейс

интуитивен , они имеют в виду, что он (a) воспринимаемый, (b) прозрачный в использовании и (c) подчиняется правилу наименьшей неожиданности[103]. Из трех данных правил правило наименьшей неожиданности является наименее ограничивающим. С первоначальной неожиданностью можно справиться, если воспринимаемость и прозрачность делают долгосрочное использование стоящим.

Пользовательские интерфейсы современных мобильных телефонов (например) характеризуются относительно высокой мнемонической нагрузкой, так как пользователю приходится сохранять в памяти хотя бы приблизительную ментальную модель интерфейсных меню, для того чтобы использовать их быстро и без необходимости постоянно уделять внимание расположению текущего пункта в иерархии. Однако хорошо спроектированные интерфейсы в любом случае быстро становятся "интуитивными" для своих пользователей, благодаря тому, что обладают всеми указанными качествами.

Понятое интуитивности не тождественно понятию простоты, поскольку (как показывает пример мобильных телефонов) пользователи способны развивать свою "интуицию" относительно прозрачных интерфейсов, имеющих довольно высокую мнемоническую нагрузку, до тех пор, пока простые операции легко осуществимы и есть способ, позволяющий быстро освоить наиболее трудные функции в интерфейсе.

11.6. Модели проектирования интерфейсов в Unix

В традициях операционной системы Unix описанные выше компромиссы достигаются с помощью твердо установившихся моделей проектирования интерфейсов. Ниже представлены лучшие из этих моделей, их анализ и примеры, после чего описывается их применение.

Следует отметить, что в число лучших примеров не включены модели GUI-конструкций (хотя включается модель проектирования, способная использовать GUI как компонент). В самих графических пользовательских интерфейсах не существует моделей проектирования,

которые были бы особенно характерными для Unix. Общее описание моделей проектирования GUI-интерфейсов можно найти в статье

"Experiences — A Pattern Language for User Interface Design" [15].

Кроме того, следует отметить, что программы могут иметь режимы, которые подходят для нескольких моделей интерфейсов. Программа, имеющая интерфейс, подобный интерфейсу компилятора, например, может работать как фильтр, если в командной строке не указаны файловые аргументы (подобным образом работают многие конвертеры форматов).

11.6.1. Модель фильтра

Моделью проектирования интерфейсов, которая наиболее традиционно связывается с операционной системой Unix, является

фильтр (filter) . Программа-фильтр принимает данные на стандартном вводе, трансформирует их определенным образом, после чего они могут запрашивать параметры начальной загрузки (например, переменные окружения), и, как правило, управляются параметрами командной строки, но не требуют обратной связи или ввода пользовательских команд во входной поток.

Двумя классическими примерами фильтров являются программы

tr(1) и

grep(1) . Программа

tr(1) представляет собой утилиту, которая преобразует данные на стандартном вводе в результаты на стандартном выводе, используя спецификацию преобразования, заданную в командной строке. Программа

grep(1) выбирает из потока стандартного ввода строки, соответствующие выражению, указанному в командной строке. Полученные в результате такого отбора строки отправляются на стандартный вывод. Третья программа-фильтр — утилита

sort(1) , которая сортирует входящие строки согласно критериям, заданным в командной строке, и отправляет отсортированный результат в поток стандартного вывода.

Обе утилиты

grep(1) и

sort(1) (но не

tr(1)) способны в качестве альтернативы принимать данные из файла (или файлов), указанного в командной строке. В таком случае программы не считывают стандартный ввод, а функционируют так, как если бы поток ввода представлял собой соединение содержимого заданных файлов в порядке следования их имен в командной строке. (В данном случае также ожидается, что ввод в командной строке символа "-" в качестве имени файла вынуждает программу считывать данные со стандартного ввода.) Прообразом таких "cat-подобных" фильтров является программа

cat(1) , причем предполагается, что фильтры функционируют именно так, если не существует

специфических для приложения причин иначе обрабатывать файлы, заданные в командной строке.

При разработке фильтров полезно учитывать несколько дополнительных правил, которые были частично сформулированы в главе 1.

1.

Помнить рекомендацию Постела: быть снисходительным к принимаемым данным и строгим к отправляемым данным. То есть пытаться принимать настолько свободные и неорганизованные входные форматы, насколько это возможно, и выпускать насколько возможно хорошо структурированный и компактный исходящий формат. Соблюдение первого условия сокращает вероятность того, что разрабатываемый фильтр будет хрупким в ситуации неожиданного применения и сломается в чужих руках (или в центре чужой инструментальной связки). Соблюдение второго условия увеличивает вероятность того, что данный фильтр будет полезен в качестве ввода для других программ.

2.

В ходе фильтрации никогда не следует отбрасывать ненужную информацию. Это также увеличивает вероятность того, что данный фильтр когда-нибудь будет полезен в качестве ввода для других программ. Отброшенная информация — информация, которую не сможет использовать ни один из последующих этапов конвейера.

3.

При фильтрации не следует добавлять излишнюю информацию. Необходимо избегать добавления несущественной информации и переформатирования, которое может усложнить анализ отфильтрованных данных в последующих программах. Наиболее распространенными примерами такой информации являются такие косметические добавления, как заголовки, колонтитулы, пустые строки и линейки, итоговые сводки и преобразования, такие как добавление выровненных столбцов или запись "150%" вместо коэффициента "1.5". Значения времени и даты — предмет особого беспокойства, поскольку их сложно анализировать в последующих программах. Любые подобные дополнения должны быть необязательными и контролироваться ключами командной строки. Если разрабатываемая программа выводит даты, то хорошей практикой является наличие ключа, который позволяет выводить их в ISO8601-форматах YYYY-MM-DD и hh:mm:ss, или, что еще лучше, использовать эти форматы по умолчанию.

Термин "фильтр" для данной модели является давно укоренившимся жаргоном Unix.

Понятие "фильтр" действительно давно укоренилось. Оно стало использоваться в то же время, что и каналы (pipes). Термин был заимствован у инженеров-электриков: данные поступали от источника через фильтры в приемник. Источником или приемником мог быть либо процесс, либо файл. Коллективный термин инженеров- электриков, "цепь" (circuit), никогда не рассматривался, так как "водопроводная метафора" для потока данных уже твердо укрепилась. Дуг Макилрой.

Некоторые программы имеют конструкторские модели, подобные фильтрам, но еще проще (и, что существенно, их проще включать в сценарий). Ими являются заклинания (cantrips), источники (sources) и приемники (sinks).

11.6.2. Модель заклинаний

Модель заклинаний является простейшей из моделей проектирования интерфейсов. Нет ни ввода, ни вывода данных, только вызов и числовой код завершения. Режим работы заклинания контролируется только условиями запуска. Не существует другого типа программ, которые проще было бы использовать в сценариях, чем заклинания.

Таким образом, модель заклинаний является замечательным стандартом в ситуациях, когда программе во время выполнения не требуется взаимодействовать с пользователем, кроме получения довольно простых установок первоначальных условий или управляющей информации.

На практике Unix-разработчики учатся противостоять соблазну написания более интерактивных программ, когда для решения задачи вполне подходят заклинания. Набором заклинаний всегда можно управлять из интерактивного упаковщика или shell-программы, но включать в сценарии интерактивные программы сложнее. Следовательно, хороший стиль программирования требует, чтобы разработчик пытался найти конструкцию заклинания для своего средства, прежде чем поддаваться соблазну написания интерактивного интерфейса, который сложнее включать в сценарии. А в ситуациях, когда интерактивность кажется обязательной, необходимо помнить характерную для операционной системы Unix модель проектирования с отделением ядра от интерфейса. Нередко верным решением является написание на каком-либо языке сценариев интерактивного упаковщика, который для выполнения реальной работы вызывает заклинание.

Консольная утилита

clear(1) , которая просто очищает экран, является заклинанием в чистейшей из возможных его форм. Данная утилита даже не принимает параметров командной строки. Другими классическими примерами являются утилиты

rm(1) и

touch(1) . Программа

startx(1) , которая применяется для запуска X-сервера, представляет собой комплексный пример, типичный для целого класса программ вызова демонов.

Несмотря на то, что данная модель проектирования является довольно распространенной, она не получила традиционного названия; термин "заклинания" (cantrips) придуман автором. (Cantrips — шотландское слово, первоначально буквально означающее магическое заклинание, которое было подобрано в популярной фантастической ролевой игре для обозначения волшебного слова, которое может быть применено немедленно с минимальной подготовкой или без подготовки.)

11.6.3. Модель источника

Источником (source) является фильтр-подобная программа, которая не требует входных данных, а ее вывод управляется только начальными условиями. Принципиальными примером является утилита

ls(1) , программа для отображения содержимого каталогов в Unix. В число других классических примеров входят программы

who(1) и ps(1) .

В операционной системе Unix генераторы отчетов, такие как

ls(1), ps(1) и

who(1) , строго руководствуются моделью источника, так что их вывод можно фильтровать с помощью стандартных инструментальных средств.

Понятие "источник" (source) является, как отмечал Дуг Макилрой, весьма традиционным. Оно менее широко распространено, чем могло бы быть, поскольку термин "источник" имеет другие важные толкования.

11.6.4. Модель приемника

Приемник (sink) — фильтр-подобная программа, которая принимает данные на стандартном вводе, но не отправляет никаких данных на стандартный вывод. Как и в двух предыдущих моделях, действия программы управляются только стартовыми условиями.

Данная модель интерфейсов необычна, и существует только несколько широко известных примеров. Одним из них является программа

lpr(1) , спулер печати в Unix, которая помещает в очередь для печати текст, переданный ей на стандартном вводе. Как и многие программы-приемники, данная утилита также обрабатывает файлы, указанные в командной строке. Другим примером является программа

mail(1) в режиме отправки почты.

Многие программы, которые на первый взгляд могут показаться приемниками, принимают управляющую информацию, как и данные на стандартном вводе, и фактически являются вариантами модели

ed (рассматривается ниже).

Понятие

"губка" (sponge) иногда применяется именно для программ-приемников, подобных утилите

sort(1) , которые должны полностью считать входные данные, прежде чем смогут обрабатывать любую их часть.

Термин "приемник" является традиционным и общепринятым.

11.6.5. Модель компилятора

Программы, подобные компиляторам, не используют ни стандартный вывод, ни стандартный ввод; однако они способны отправлять сообщения об ошибках в соответствующий поток данных (stderr). Вместо этого программы данного типа принимают имена файлов или ресурсов из командной строки, определенным образом преобразовывают имена данных ресурсов и отправляют вывод в файлы с трансформированными именами. Как и заклинания,

программы, подобные компиляторам, после запуска не нуждаются во взаимодействии с пользователем.

Данная модель названа так потому, что основным образцом для нее служит компилятор языка C,

cc(1) (или

gcc(1) в Linux и многих других современных Unix-системах). Однако данная модель также широко применяется для программ, осуществляющих (например) преобразование или компрессию/декомпрессию графических файлов.

Хорошим примером программ-конвертеров является утилита

gif2png(1) , которая применяется для преобразования формата GIF (Graphic Interchange Format) в PNG (Portable Network Graphics)[104]. Хорошими примерами программ компрессии/декомпрессии являются GNU-утилиты

gzip(1) и

gunzip(1) , несомненно, поставляемые с большинством Unix-систем.

Как правило, модель компилятора хорошо применима при проектировании конструкции интерфейса, когда разрабатываемой программе часто приходится оперировать с множеством именованных ресурсов и ее можно написать так, чтобы она не требовала высокой интерактивности (т.е. когда управляющая информация предоставляется во время запуска программы). Программы, подобные компиляторам, можно легко включать в сценарии.

Понятие "компиляторный интерфейс" (compiler-like interface) для данной модели понятно многим в Unix-сообществе.

11.6.6. Модель редактора

ed

Для всех предыдущих моделей характерна весьма низкая интерактивность. В них используется только управляющая информация, переданная во время запуска и обособленная от данных. Однако многим программам после запуска требуется управление с помощью продолжительного диалога с пользователем.

Традиционно для Unix, простейшая модель проектирования интерактивного интерфейса иллюстрируется на примере строчного редактора

ed(1) . В число других классических примеров включаются

ftp(1) и

sh(1) , оболочка Unix. Программа

ed(1) принимает в качестве аргумента имя файла и модифицирует данный файл. На входе программа принимает командные строки. Некоторые из команд отражаются на выходных данных на стандартном выводе для немедленного просмотра пользователем, как часть диалога с программой.

Реальный пример сеанса работы в редакторе

ed(1) включен в главу 13.

Многие программы в Unix, подобные браузерам и редакторам, придерживаются данной модели, даже когда редактируемые ими именованные ресурсы не являются текстовыми файлами. В качестве примера можно упомянуть символьный GNU-отладчик

Программы, подчиняющиеся модели

ed , не так широко можно использовать в сценариях, как можно было бы использовать более простые типы интерфейсов, аналогичные фильтрам. Таким программам можно передать команды через стандартный ввод, но генерировать последовательности команд (и интерпретировать любой их вывод) сложнее, чем просто устанавливать значения переменных окружения и параметры командной строки. Если действие команд не является настолько предсказуемым, что они могут выполняться вслепую (например, с потоковым документом (here document) в качестве входных данных и игнорируя вывод), то управление ed-подобными программами требует протокола и соответствующего конечного автомата в вызывающем процессе. Это приводит к проблемам, отмеченным в главе 7 при обсуждении управления подчиненными процессами.

Тем не менее, данная модель является простейшей и предоставляет наибольшие возможности использования в сценариях, которые доступны для полностью интерактивных программ. Соответственно, она остается весьма полезной как компонент модели "разделения ядра и интерфейса", которая описывается ниже.

11.6.7. Rogue-подобная модель

Данная модель получила свое название благодаря первому примеру ее реализации, ролевой игре

rogue(1) (см. рис. 11.2) в BSD. Определение "rogue-подобная" для данной модели широко известно в традиции Unix. Rogue-подобные программы предназначены для запуска в системной консоли, эмуляторе X-терминала или видеотерминале. Они используют полный экран и поддерживают визуальный стиль интерфейса, но с псевдографическим экраном вместо графического экрана и мыши.

Команды обычно представлены в виде одиночных клавиш, ввод которых не отображается для пользователя (в противоположность командным строкам ed-модели), хотя некоторые из них приводят к открытию командного окна (часто, хотя и не всегда, последняя строка экрана), в котором можно вводить более сложные вызовы. Структура команд заставляет пользователя интенсивно использовать клавиши управления курсором для выбора участков экрана или строк, на которых необходимо выполнить какую-либо операцию.

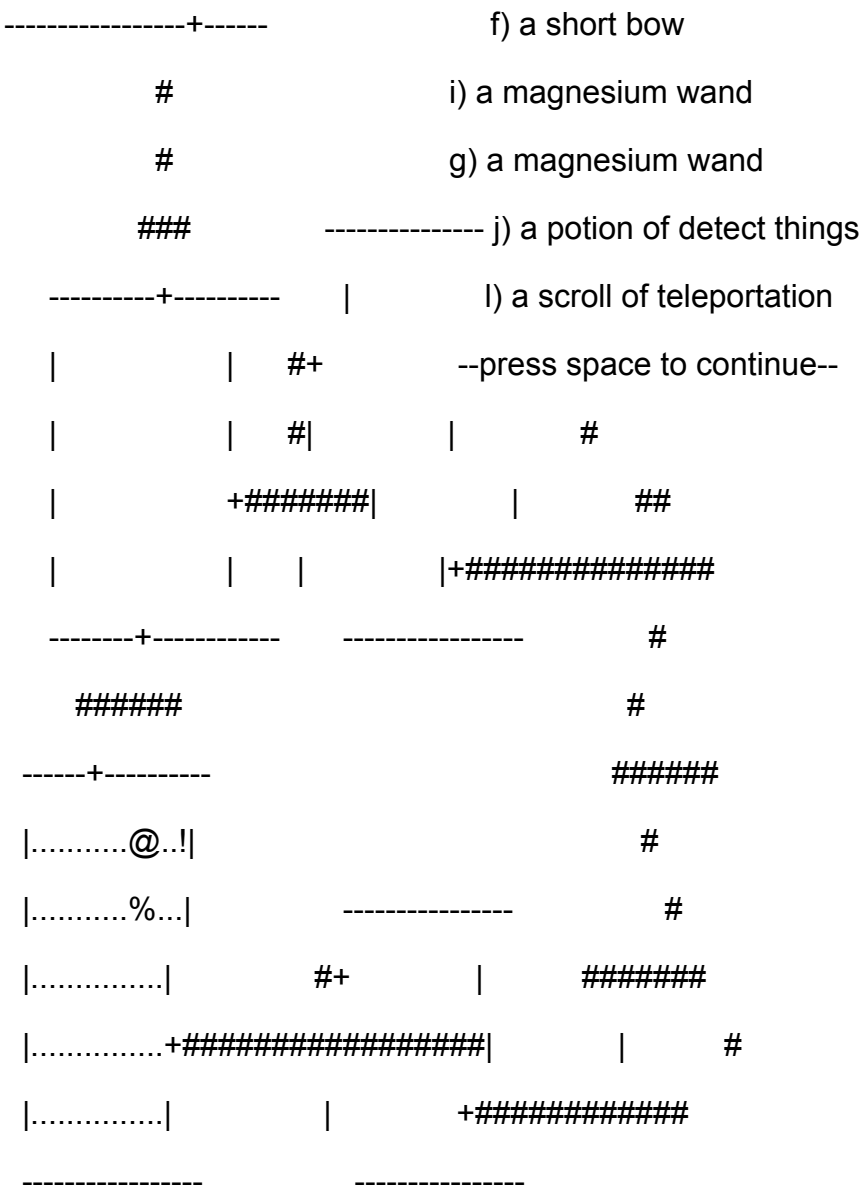
a) some food

b) +1 ring mail [4] being worn

---------- ########## c) a +1, +2 mace in hand

| +################ d) a +1,+0 short bow

| | e) 28 +0,+0 arrows



Level: 3 Gold: 73 Hp: 36(36) Str: 14(16) Arm: 4 Exp: 4/78

Рис. 11.2. Копия экрана оригинальной игры Rogue

Программы, написанные в данном стиле, часто моделируются на основе редакторов

vi(1) либо

emacs(1) и (подчиняясь правилу наименьшей неожиданности) используют их последовательности команд для выполнения таких частых операций, как получение справки и завершение программы. Следовательно, можно, например, ожидать, что программа, написанная согласно данной модели, завершит работу после команд "x", "q" или "C-x C-c".

В число других элементов интерфейсов, связываемых с данной моделью, входят: (a) меню, каждая строка которого состоит из одного пункта, причем текущий выбор выделяется жирным шрифтом или инверсной подсветкой, и (b) "строки состояния" — сводные данные о состояния программы, представленные в подсвечиваемой строке экрана часто вблизи нижней или верхней его границы.

Rogue-подобная модель развивалась в окружении видеотерминалов. Многие из них не имели клавиш-стрелок или функциональных клавиш. В мире персональных компьютеров, поддерживающих графические интерфейсы, с постепенно исчезающими из памяти псевдографическими терминалами, не трудно забыть, какое влияние данная модель оказала

на проектирование. Но ранние экземпляры rogue-подобной модели были разработаны за несколько лет до того, как в 1981 году IBM стандартизировала клавиатуру персонального компьютера. Как следствие, традиционной, но в настоящее время архаичной частью модели является использование клавиш h, j, k и l в качестве клавиш управления курсором, когда они не интерпретируются как самостоятельные символы в окне редактирования. Неизменно клавиша k соответствует стрелке "вверх", j — стрелке "вниз", h — стрелке "влево" и l — стрелке "вправо". Данный исторический факт также объясняет, почему в старых Unix-программах часто не использовались клавиши ALT, а функциональные клавиши использовались ограниченно или вообще не использовались.

Существует множество программ, подчиняющихся данной модели: текстовый редактор

vi(1) во всех его вариантах, а также редактор

emacs(1); elm(1), pine(1), mutt(1) и большинство других Unix-программ для чтения почты;

tin(1), slrn(1) и другие программы чтения новостей в Unix; Web-браузер

lynx(1) и многие др. Большинство Unix-программистов в основном управляют программами с подобными интерфейсами.

Программы Rogue-подобной модели трудно включать в сценарии. На практике такой подход редко встречается. Среди прочего в данной модели используется режим непосредственного, посимвольного ввода, который неудобен для написания сценариев. Также весьма затруднительно программно интерпретировать вывод, поскольку он обычно состоит из последовательностей действий по окрашиванию экрана.

Также в данной модели отсутствует визуальная гладкость, присущая управляемому с помощью мыши полному GUI-интерфейсу. Несмотря на то, что целью использования полноэкранного интерфейса является поддержка простых видов непосредственного управления и интерфейсов на основе меню, в rogue-подобных программах до сих пор требуется, чтобы пользователи изучали весь ассортимент команд. Действительно, интерфейсы, созданные на основе данной модели, демонстрируют тенденцию к нагромождению режимов и команд, причем требующих одновременного нажатия нескольких клавиш, что может удовлетворить только радикально настроенных хакеров. Может показаться, что данная модель приобрела худшие качества обоих подходов: она не предоставляет возможности включения в сценарии и не согласуется с последними течениями в дизайне интерфейсов для конечных пользователей.

Однако в данной модели существует и определенная ценность. Rogue-подобные почтовые программы, программы для чтения новостей, редакторы и другие программы остаются чрезвычайно популярными даже среди пользователей, которые неизменно запускают их посредством эмуляторов терминала на X-дисплее, поддерживающем GUI-конкурентов. Более того, rogue-подобная модель настолько распространена, что в операционной системе Unix даже GUI-программы часто подражают ей, добавляя поддержку мыши и графики в командный и экранный интерфейс, который выглядит скорее rogue-подобно. Характерными примерами такой доработки являются программы

emacs(1) и клиент

xchat(1) . В чем причина неисчезающей популярности данной модели?

Важными факторами представляются эффективность и ощутимая польза. Для rogue-подобных программ характерна более высокая скорость работы и легковесность по сравнению с их ближайшими GUI-конкурентами. С точки зрения скорости запуска и выполнения, использование rogue-подобной программы в Xterm может быть более

предпочтительным по сравнению с запуском GUI-интерфейса, который потребляет солидные ресурсы, устанавливая параметры дисплея и впоследствии медленнее отвечая на запросы пользователя. Кроме того, программы на основе рассматриваемой модели могут применяться на telnet-каналах или низкоскоростных коммутируемых соединениях, для которых X-средства недоступны.

Владеющие "слепым" методом набора операторы предпочитают rogue-подобные программы, поскольку, работая с ними не нужно снимать руки с клавиатуры и перемещать мышь. Имея выбор, такие операторы предпочтут интерфейсы, сокращающие нажатия клавиш, находящихся выше клавиши "home". Возможно, это наибольшая весомая причина популярности редактора

vi(1) .

Но, более важно то, что rogue-подобные интерфейсы являются предсказуемыми и экономно используют площадь X-дисплея. Они не загромождают дисплей множеством окон, элементами управления, диалоговыми окнами и другим GUI-снаряжением. Это делает данную модель подходящей для использования в программах, которые совместно с другими программами должны часто разделять внимание пользователя (это особенно характерно для редакторов, почтовых программам, программам чтения новостей, chat-клиентов и других коммуникационных программ).

Наконец (и, вероятно, это наиболее важно), rogue-подобная модель часто более, чем GUI, привлекает пользователей, которые ценят лаконичность и выразительность командного набора достаточно высоко для того, чтобы принять дополнительную мнемоническую нагрузку. Выше отмечалось, что существуют весомые причины для того, чтобы такой подход становился более распространенным по мере роста сложности задач, частоты использования и опыта пользователей. Данная модель соответствует предпочтениям таких пользователей, одновременно поддерживая GUI-подобные элементы непосредственного управления, которые не способна поддерживать

ed- модель. Таким образом, rogue-подобная модель имеет не только худшие свойства обоих типов интерфейса, но и способна собрать в себе некоторые лучшие их черты.

11.6.8. Модель "разделения ядра и интерфейса"

В главе 7 рассматривались доводы против создания крупных однопроцессных монолитов, а также обсуждалась возможность понижения глобальной сложности программ путем их разделения на взаимодействующие блоки. В мире Unix данная тактика часто реализуется путем отделения "ядра" программы (основных алгоритмов и логики, специфической для данной прикладной области) от "интерфейсной" ее части (которая принимает пользовательские команды, отображает результаты, а также может предоставлять такие службы, как интерактивная справочная система или история команд). В сущности, данная модель обособленного ядра и интерфейса, вероятно, является наиболее характерной моделью проектирования интерфейсов в Unix.

Другими более очевидными кандидатами для такого разграничения были бы фильтры. Однако фильтры в не-Unix-средах встречаются чаще, чем пары ядро/интерфейс с двунаправленной передачей данных между ними. Имитировать конвейеры просто. Сложно реализовать более развитые IPC-механизмы, необходимые для пар ядро/интерфейс.

Оуэн Тейлор (Owen Taylor), куратор библиотеки GTK+, широко используемой для создания

пользовательских интерфейсов в системе X, превосходно показал технические преимущества данного вида разделения в конце своей заметки "Why GTK_MODULES is not a security hole" <http://www.gtk.org/setuid.html>. Он заканчивает статью так: "Безопасной setuid-программой является программа в 500 строк кода, выполняющая только то, что необходимо, а не библиотека в 500 000 строк, основной задачей которой является поддержка пользовательских интерфейсов".

Эта идея не нова. Ранние исследования сотрудников центра Xerox PARC в области графических пользовательских интерфейсов привели их к созданию схемы "модель-отображение-контроллер" (model-view-controller) как прототипа для GUI-интерфейсов.

• "Модель" в данном случае — то, что в мире Unix обычно называется "ядром" (engine). Модель содержит структуры данных, зависящие от предметной области, и логику приложения. Серверы баз данных являются основными примерами моделей.

• В часть "отображение" входит то, что приводит объекты предметной области в визуальную форму. В приложении с действительно четко разделенными "моделью/отображением/контроллером" компонент отображения извещается об обновлениях модели и отвечает скорее самостоятельно, а не под синхронным управлением контроллера или явных запросов на обновление.

• Компонент "контроллер" обрабатывает пользовательские запросы и передает их модели в виде команд.

На практике части "отображение" и "контроллер" часто сильнее связаны друг с другом, чем с моделью. Например, в большинстве GUI-интерфейсов комбинируется поведение частей "отображения" и "контроллера". Они разделяются только в тех случаях, когда приложение требует нескольких видов отображения модели.

Схема модель/отображение/контроллер в операционной системе Unix распространена гораздо шире, чем в других средах, именно потому, что существует прочная традиция "решать одну задачу хорошо", а IPC-методы являются одновременно простыми и гибкими.

Особенно мощная форма данной методики связывает интерфейс политики (часто GUI-интерфейс, комбинирующий функции отображения и контроллера) с ядром (моделью), который содержит интерпретатор для узкоспециального мини-языка. Данная модель проектирования рассматривалась в главе 8, где основное внимание было уделено конструкциям мини-языков. Здесь рассматриваются различные способы, с помощью которых такие ядра могут формировать компоненты более крупных систем кода.

Существует несколько основных вариантов данной модели проектирования.

11.6.8.1. Пара конфигуратор/актор

В данной паре интерфейсная часть управляет средой запуска фильтра или программы, подобной демону, которая затем выполняется, не требуя пользовательских команд.

Хорошим примером пары конфигуратор/актор являются программы

fetchmail(1) и

fetchmaileonf(1) (которые уже использовались как учебные примеры воспринимаемости и

создания программ, управляемых данными, а также рассматриваются в главе 14 как учебные примеры использования языка Python). Утилита

fetchmailconf представляет собой интерактивный конфигуратор файлов профилей, который поставляется с программой fetchmail.

fetchmailconf также может служить в качестве GUI-упаковщика, который запускает fetchmail либо в приоритетном, либо в фоновом режиме.

Данная модель проектирования позволяет обеим программам,

fetchmail и

fetchmailconf , специализироваться в том, что они делают хорошо, и конечно, позволяет писать их на различных языках, которые целесообразно использовать в данных предметных областях. Программу fetchmail, которая обычно запускается в фоновом режиме как демон, не требуется объединять с GUI-кодом, и наоборот,

fetchmailconf может специализироваться на сложном GUI-представлении, не требуя от fetchmail затрат размера и сложности. Наконец, ввиду того, что информационные каналы между ними являются узкими и четко определенными, остается возможность управлять программой

fetchmail из командной строки, а также из сценариев, отличных от

fetchmailconf .

Термин "конфигуратор/актор" придуман автором данной книги.

11.6.8.2. Пара спулер/демон

Облегченный вариант пары конфигуратор/актор может оказаться полезным в ситуациях, когда требуется сериализованный доступ к общему ресурсу в пакетном режиме, т.е. когда четко определенный поток заданий или последовательность запросов требует некоторого совместно используемого ресурса, но ни одна отдельная задача не требует взаимодействия с пользователем.

В данной модели спулер или клиентская часть просто помещает запросы на выполнение заданий и данные в спул-область. Запросы на выполнение заданий и данные — просто файлы, а спул-областью, как правило, является каталог. Расположение данного каталога и формат запросов согласовывается между спулером и демоном.

Демон постоянно работает в фоновом режиме, опрашивая спул-каталог в поисках задания. Когда он находит запрос на выполнение задания, то пытается обработать связанные с ним данные. Если обработка прошла успешно, то запрос и данные из спул-области удаляются.

Классическим примером данной модели является система спулера печати Unix,

lpr(1)/lpd(1) . Интерфейсной частью в данном случае является утилита

lpr(1) , просто помещающая предназначенные для печати файлы в спул-область, которая периодически сканируется демоном

lpd . Задача данного демона — сериализация доступа к печатающим устройствам.

Другим классическим примером является пара

at(1)/atd(1) , обеспечивающая выполнение команд по расписанию. Третьим исторически важным примером, хотя и не распространенным в настоящее время, была программа UUCP (Unix-to-Unix Copy Program), широко применявшаяся в качестве почтового транспорта на коммутируемых линиях до того, как в начале 90-х годов прошлого века не начался взрывной рост Internet.

Модель спулер/демон и сейчас остается важной в программах транспортировки почты (которые по своей природе являются пакетными). Интерфейсные части почтовых транспортных программ, таких как

sendmail(1) и

qmail(1), обычно предпринимают одну попытку немедленной доставки почты через SMTP-сервер и внешнее Internet-соединение. Если попытка не удалась, то почта помещается в спул-область; демон-версия или режим почтового транспорта попытается доставить почту позднее.

Как правило, система спулер/демон состоит из четырех частей: модуль запуска задания, организатор очереди, утилита отмены задания и спулер-демон. Фактически присутствие первых трех частей определенно указывает на то, что за ними находится спулер-демон.

Понятия "спулер" и "демон" — прочно укоренившийся жаргон в Unix-сообществе (фактически термин "спулер" появился во время ранних мэйнфреймов).

11.6.8.3. Пара драйвер/ядро

В данной модели, в отличие от пары конфигуратор/актор или спулер/демон, интерфейсная часть подает команды и интерпретирует вывод от ядра после запуска. Ядро имеет простую модель интерфейса. Используемый IPC-метод является деталью реализации: ядро может быть подчиненным процессом драйвера (в том смысле, который обсуждался в главе 7), или ядро и драйвер могут взаимодействовать через сокеты, общую память, или посредством любого другого IPC-метода. Ключевыми моментами в данном случае являются (a) интерактивность пары и (b) способность ядра к самостоятельной работе со своим собственным интерфейсом.

Написание таких пар сложнее, чем написание пар конфигуратор/актор, поскольку они теснее и сложнее связаны. Драйверу необходимы сведения не только об ожидаемой среде запуска ядра, но и о его наборе команд, а также о форматах ответов.

Однако если ядро спроектировано с возможностью использования в сценариях, то нередко драйверная часть пишется не автором ядра, а другим разработчиком, или несколько драйверов могут подключаться к определенному ядру. Превосходный пример в обоих случаях представлен программами

gv(1) и ghostview(1) , которые являются драйверами для

gs(1) , интерпретатора Ghostscript. Ghostscript переводит PostScript в различные графические форматы и низкоуровневые языки управления принтерами. Программы gv и ghostview обеспечивают GUI-упаковщики для весьма уникальных ключей вызова и синтаксиса команд Ghostscript.

Другим превосходным примером данной модели является комбинация программ xcdroast/cdrtools. В дистрибутив cdrtools входит программа

cdrecord(1) с интерфейсом командной строки. Код

cdrecord специализируется на работе с аппаратным обеспечением приводов CD-ROM. Программа

xcdroast представляет собой GUI-интерфейс и предназначена для того, чтобы создать комфортные условия работы для пользователя. Для выполнения большей части работы программа

xcdroast(1) вызывает

cdrecord(1) .

Программа

xcdroast также вызывает другие CLI-инструменты:

cdda2wav(1) (конвертер звуковых файлов) и

mkisofs(1) (средство для создания из списка файлов CD-ROM-образов с файловой системой ISO-9660). Подробности того, как данные средства вызываются, скрыты от пользователей, которые могут сконцентрировать внимание на создании CD-образов, а не на необходимости знать секреты преобразования звуковых файлов или структуры файловых систем. Равно важно и то, что разработчики каждого из данных средств могут концентрироваться на профессиональных знаниях в своей предметной области без необходимости быть экспертами в разработке пользовательских интерфейсов.

Главный "подводный камень" организации драйвер/ядро состоит в том, что часто драйвер должен распознавать состояние ядра, для того чтобы отражать его пользователю. Если ядро работает практически без задержек, то это не является проблемой, однако если ядру требуется большой промежуток времени (например, при доступе к множеству URL-адресов), то недостаток обратной связи может создавать значительные сложности. Подобной проблемой является отклик на ошибки. Например, традиционный (хотя и не совсем соответствующий духу Unix) запрос на подтверждение при перезаписи существующего файла сложен в написании в системе драйвер/ядро. Ядру, обнаружившему проблему, приходится требовать от драйвера выдать сообщение пользователю. Стив Джонсон.

Важно спроектировать ядро так, чтобы оно не только правильно работало, но и уведомляло драйвер о выполняемой работе, с тем чтобы драйвер мог представить изящный интерфейс с соответствующей обратной связью.

Понятия "драйвер" и "ядро" являются редкими, но устоявшимися в Unix-сообществе.

11.6.8.4. Пара клиент/сервер

Пара клиент/сервер подобна паре драйвер/ядро, за исключением того, что часть ядра является запущенным в фоновом режиме демоном, от которого не требуется интерактивная работа и наличие собственного пользовательского интерфейса. Обычно демон предназначен для того, чтобы быть посредником в доступе к какому-либо совместно используемому ресурсу — базе данных, потоку транзакций или совместно используемому специализированному

аппаратному обеспечению, такому как звуковое устройство. Другой причиной для использования такого демона может быть необходимость избежать выполнения дорогостоящих начальных действий при каждом вызове программы.

Рис. 11.3. GUI-интерфейс программы Xcdroast

Раньше, например, очень популярной была пара

ftp(1)/ftpd(1) , реализующая протокол FTP (File Transfer Protocol — протокол передачи файлов); также в качестве примера можно упомянуть два экземпляра программы

sendmail(1) — это отправитель в приоритетном режиме и слушатель в фоновом, передающие электронную почту в Internet. В настоящее время таким примером будет любая пара браузер/Web-сервер.

Однако данная модель не ограничена коммуникационными программами. Другим важным случаем ее применения являются базы данных, например пара

psql(1)/postmaster(1) . В данной паре программа

psql сериализует доступ к совместно используемой базе данных, управляемой демоном postgres, передавая ему SQL-запросы и представляя данные, отправленные в качестве ответов.

Приведенные выше примеры иллюстрируют важное свойство данных пар, которое заключается в том, что чистота протокола, сериализующего связь между ними, является крайне важной. Если протокол четко определен и описан открытым стандартом, то он может предоставить широкие возможности для развития, изолируя клиентские программы от деталей управления серверными ресурсами и позволяя клиентам и серверам развиваться почти независимо. Все программы с ядром, отделенным от интерфейса, потенциально извлекают данное преимущество из четкого разделения функций, но в случае клиент/сервер выигрыш от верной реализации часто особенно высок именно потому, что управлять общими ресурсами действительно сложно.

Для связи клиентской части с серверной могут использоваться и используются очереди сообщений и пары именованных каналов, однако преимущества от возможности запуска сервера на другой машине являются настолько весомыми, что в настоящее время почти во всех современных парах клиент/сервер используются TCP/IP-сокеты.

11.6.9. Модель CLI-сервера

В мире Unix обычной является практика, когда серверные процессы вызываются управляющими программами[105], такими как

inetd(8) , т.е. сервер может получать команды через стандартный ввод и отправлять ответы на стандартный вывод. Управляющая программа затем обеспечивает подключение стандартного ввода и вывода сервера к порту определенной TCP/IP-службы. Преимущество такой организации функций заключается в том, что управляющая программа может работать как одна безопасная программа-диспетчер (gatekeeper) для всех серверов, которые она запускает.

Таким образом, одной из классических моделей интерфейсов является CLI-сервер. CLI-сервер — программа, которая при запуске в привилегированном режиме имеет простой

CLI-интерфейс, считывающий данные со стандартного ввода и отправляющий данные на стандартный вывод. Когда сервер запускается в фоновом режиме сервер, он обнаруживает это и подключает свой стандартный ввод и вывод к определенному TCP/IP-порту.

В некоторых вариантах данной модели сервер по умолчанию переводится в фоновый режим, а для того чтобы он оставался в привилегированном режиме, в командной строке необходимо использовать определенный ключ. Но еще более важно то, что большая часть кода "не знает" и "не заботится" о том, работает ли программа в привилегированном режиме или запущена как управляющая TCP/IP-программа.

Серверы POP3, IMAP, SMTP и HTTP обычно соответствуют данной модели. Ее можно комбинировать с любыми моделями клиент/сервер, описанными ранее в данной главе. HTTP-сервер также может функционировать в качестве управляющей программы. CGI-сценарии, которые создают большую часть интерактивного содержания в Web, выполняются в обеспеченной данным сервером специальной среде, где они могут принимать ввод (из аргументов) из стандартного ввода и записывать сгенерированные в результате своей работы HTML-страницы в стандартный вывод.

Хотя данная модель весьма традиционна, термин "CLI-сервер" впервые введен в употребление автором данной книги.

11.6.10. Модель интерфейсов на основе языков

В главе 8 рассматривались узкоспециальные мини-языки как средства перемещения спецификации программ на более высокий уровень с приобретением гибкости и сокращением количества ошибок. Благодаря этим достоинствам CLI-интерфейс на основе языков сегодня во многом определяет стиль Unix-интерфейсов.

Сильные стороны данной модели хорошо иллюстрируются приведенным ранее в данной главе учебным примером, в котором комбинация утилит

dc(1)/bc(1) сравнивается с программой

xcalc(1) . Преимущества, отмеченные ранее (выигрыш в выразительности и возможностях использования в сценариях), типичны для мини- языков. Они обобщаются в других ситуациях, в которых обычно приходится последовательно выполнять сложные операции в специализированной прикладной области. Довольно часто, в отличие от примера с калькулятором, мини-языки также обладают явным преимуществом лаконичности.

Одной из наиболее мощных моделей проектирования в Unix является комбинация клиентского GUI-интерфейса с серверной частью, представленной CLI-мини-языком. Хорошо спроектированные примеры данного типа неизбежно сложны, но часто являются более простыми и гибкими, чем большой объем уникального кода, который понадобился бы для решения даже небольшой части тех задач, которые способен решить мини-язык.

Данная общая модель, без сомнения, не является уникальной для Unix. Современные пакеты приложений для работы с базами данных в большинстве операционных систем обычно состоят из одного или нескольких GUI-клиентов и генераторов отчетов, каждый из которых обращается к общему серверу, используя язык запросов, такой как SQL. Однако данная модель развивалась преимущественно в Unix-среде и до сих пор остается в ней гораздо более понятной и применяется шире, чем в каких-либо других операционных системах.

Когда клиентская и серверная части системы, удовлетворяющей данной модели проектирования, скомбинированы в одной программе, то о такой программе часто говорят, что она имеет "встроенный язык сценариев". В Unix-среде одним из наиболее известных экземпляров данной модели является редактор

Emacs . Некоторые преимущества

Emacs рассматриваются в главе 8.

Другим хорошим примером является средство script-fu программы GIMP. GIMP — мощный графический редактор с открытым исходным кодом, имеющий GUI-интерфейс, подобный интерфейсу программы Adobe Photoshop. Средство script-fu позволяет использовать функции GIMP в сценариях, написанных на языке Scheme (диалект Lisp). Доступно также написание сценариев на языках Tcl, Perl или Python. Программы, написанные на любом из указанных языков, могут вызывать внутренние функции GIMP через его интерфейс подключаемых подпрограмм. Демонстрационное приложение для данного средства представлено на Web-странице[106], позволяющей создавать простые логотипы и графические кнопки посредством CGI-интерфейса, который передает сгенерированную Scheme-программу экземпляру GIMP и возвращает завершенное изображение.

11.7. Применение Unix-моделей проектирования интерфейсов

Для того чтобы облегчить написание сценариев и создание конвейеров, разумно выбрать простейшую из возможных моделей интерфейса, т.е. модель с минимальным числом каналов к окружению и наименьшей интерактивностью.

При описании многих однокомпонентных моделей подчеркивалось, что соответствующая программа после запуска не требует взаимодействия с пользователем. Если ожидается, что "пользователем" часто будет другая программа (которой, естественно, не достает гибкости человеческого мозга), то это является очень ценной особенностью, максимальной увеличивающей возможность использования программы в сценариях.

Выше уже говорилось, что в различных моделях оптимизированы характеристики, ценные в различных обстоятельствах. В частности, существует жесткое и давнее противоречие между GUI-интерфейсами, а также моделями, подходящими для начинающих и нетехнических конечных пользователей (с одной стороны), и интерфейсами, которые обслуживают опытных пользователей и максимально расширяют возможности использования программ в сценариях (с другой стороны).

Один из способов обойти эту дилемму заключается в создании программ с различными режимами, представляющими несколько моделей. В данном случае замечательным примером является Web-браузер

lynx(1) . Обычно

lynx имеет rogue-подобный интерфейс для интерактивного использования, но может быть вызван с параметром -dump, превращающим его в программу-источник, форматирующую указанную Web-страницу в текст, который распечатывается на стандартный вывод.

Однако подобные двухрежимные интерфейсы обычно не применяются, когда программа должна иметь действительно графический пользовательский интерфейс. Причины этого частично являются историческими, но главная причина — необходимость управлять

глобальной сложностью. GUI-интерфейсы склонны требовать сложной начальной конфигурации и большого количества специализированного кода. Наличие данных особенностей в более простых моделях небезопасно. В худшем случае двухрежимная программа (работающая как с GUI-интерфейсом, так и без него) может потребовать двух отдельных циклов интерпретации команд, а также выполнения других условий, подразумевающих распухание кода и потенциальную несовместимость.

Таким образом, когда выбор "простейшей модели" противоречит созданию GUI-интерфейса, Unix-способом будет разделение программы на две части согласно модели "разделение ядра и интерфейса".

В сущности, комбинируя идею из главы 7 с описанной выше, можно определить новую модель проектирования, возникшую в Linux и других современных Unix-системах с открытым исходным кодом. В этой модели GUI-интерфейсы являются не просто вынужденными дополнениями, а определяют перспективное направление для приложения усилий многих разработчиков.

11.7.1. Модель многопараметрических программ

Многопараметрическая (polyvalent) программа обладает рядом характерных особенностей.

1. Прикладная логика программы находится в библиотеке с документированным API-интерфейсом, который может быть связан с другими программами. Интерфейсная логика программы для связи с внешним миром представляет собой тонкий уровень над библиотекой. Или, возможно, существует несколько уровней с различными стилями пользовательского интерфейса, к любому из которых можно подключить данную библиотеку.

2. Одним из режимов пользовательского интерфейса является модель "заклинания", модель компилятора" или CLI-модель", в которой интерактивные команды выполняются в режиме пакетной обработки.

3. Одним из режимов пользовательского интерфейса является GUI-интерфейс, либо связанный непосредственно с основной библиотекой, либо функционирующий как отдельный процесс, управляющий CLI-интерфейсом.

4. Одним из режимов пользовательского интерфейса является интерфейс сценариев, использующий один из современных универсальных языков сценариев, например Perl, Python или Tcl.

5. Не обязательное дополнение: одним из режимов пользовательского интерфейса является rogue-подобный интерфейс, в котором используется библиотека

curses(3) .

В наибольшей степени данной модели в настоящее время соответствует пакет GIMP.

11.8. Использование Web-браузера в качестве универсального клиента

Отделение CLI-сервера от GUI-интерфейса стало особенно привлекательной стратегией

после того, как в середине 1990-х годов технология World Wide Web преобразила мир компьютерных вычислений. Для большого класса приложений возрастает смысл вообще не писать нестандартные GUI-клиенты, а вместо этого в данной роли использовать Web-браузеры.

Такой подход имеет целый ряд преимуществ. Наиболее очевидным является то, что разработчику не требуется писать процедурный GUI-код. Вместо этого можно описать желаемый GUI-интерфейс на специально предназначенных для этого языках (HTML и JavaScript). Это позволяет избежать большого количества дорогого и сложного одноцелевого кодирования и часто более чем вдвое сокращает общие усилия по проекту. Другое преимущество заключается в том, что разрабатываемое приложение немедленно становится готовым для использования в Internet. Клиент может быть расположен на том же узле, что и сервер, или находиться за тысячи километров от него. Также преимуществом является то, что все второстепенные детали представления приложения (такие как шрифты и цвет) более не являются задачей сервера, и пользователи действительно смогут выбирать их по своему усмотрению с помощью таких механизмов, как настройки браузера и каскадные таблицы стилей. Наконец, единообразные элементы Web-интерфейса существенно облегчают пользователям задачу изучения.

Данный подход имеет и недостатки. Два наиболее важных из них: (a) пакетный стиль взаимодействия, навязанный Web-средой, и (b) сложности управления постоянными сеансами при использовании протокола без фиксации состояния. Хотя указанные недостатки не являются исключительно проблемами Unix, их следует рассмотреть здесь, поскольку на уровне

конструкции весьма важно четко определить, когда стоит принять или обработать данные ограничения.

Рис. 11.4. Связи вызывающих и вызываемых программ в многопараметрической программе

Интерфейс общего шлюза (Common Gateway Interface — CGI), посредством которого браузер может вызывать программы на серверном узле, не вполне поддерживает интерактивность на уровне мелких модулей. Не поддерживают ее и шаблонные системы, серверы приложений и встроенные серверные сценарии, которые постепенно заменяют CGI (слегка злоупотребляя данным понятием, автор далее в настоящем разделе использует аббревиатуру CGI для обозначения всех указанных технологий).

Через CGI-шлюз невозможно осуществлять посимвольный или пожестовый ввод-вывод в GUI. Вместо этого необходимо заполнить HTML-форму и нажать кнопку "отправить", после чего содержимое формы будет отослано CGI-сценарию, который затем обработает данные, а сервер передаст обратно клиенту сгенерированную сценарием HTML-страницу (которая сама по себе может быть другой CGI-формой).

Описанная выше методика по существу представляет пакетный стиль взаимодействия, который концептуально недалеко ушел от загрузки перфокарт во входной бункер и получения распечатки. Этот стиль можно преобразовать в более приемлемую форму, используя для взаимодействия с пользователем JavaScript-сценарий, группирующий транзакции в сообщения для отправки серверу.

Java-аплеты способны открывать собственные соединения для символьных потоков обратно к серверу с целью поддержать более управляемую интерактивность. Однако Java имеет технические проблемы — данный язык может использовать только фиксированную область экрана на странице и не способен изменить часть экрана за пределами этого прямоугольника. Имеются также гораздо более серьезные проблемы, связанные с политикой (частное лицензирование со стороны Sun заблокировало Java-разработку и вынудило

остальных с недоверием относиться к Java; разработчик не может рассчитывать на то, что браузеры всех пользователей поддерживают данную технологию).

Кроме того, как Java, так и JavaScript могут столкнуться с несовместимостью браузеров. Сопротивление Microsoft внедрению JDK 1.2 и Swing в Internet Explorer является серьезной проблемой для Java-аплетов. Различные уровни версий JavaScript также могут нарушить работу приложения (хотя ошибки JavaScript исправляются проще). Тем не менее, часто решить данные проблемы не так сложно, как написать и внедрить нестандартный клиентский интерфейс. Более сложной проблемой является растущее число искушенных пользователей, которые по привычке отключают в своих браузерах поддержку Java и даже JavaScript из-за проблем с безопасностью и нарушения интерфейса.

Возникает также отдельная проблема — сложно поддерживать информацию о сеансах между несколькими CGI-формами. Между CGI-транзакциями сервер не хранит сведения о состоянии клиентских сеансов, поэтому на них нельзя полагаться, связывая формы, отправленные пользователем позднее, с ранее отправленными тем же пользователем формами. Существует два стандартных технических приема для решения данной проблемы: связанные формы и cookie-файлы.

При связывании форм подготавливается CGI-сценарий для формирования первой формой уникального идентификатора в скрытом поле второй формы, а вторая и все последующие формы передают данный идентификатор последующим элементам. Cookie-файлы достигают подобного эффекта путем аналогичным переменным окружения (подробные инструкции описаны в любой из сотен книг по CGI-программированию). В любом случае в CGI-сценарии необходимо использовать идентификатор в качестве индекса сеанса (или cookie-файлов для непосредственного сохранения состояния) и явно обрабатывать мультиплексирование сеансов.

Часто данными ограничениями можно пренебречь. Многие нетривиальные приложения могут уместиться в одной форме и работать, избегая обеих проблем. Даже если это не так и приложению требуется множество форм, понижение сложности и сокращение количества средств по сравнению с необходимостью создания и распространения специализированного клиента настолько велико, что может свободно окупить усилия, необходимые для написания достаточно развитых CGI-сценариев, самостоятельно осуществляющих отслеживание сеансов.

Проблема управления сеансами может быть переадресована серверам приложений, таким как Zope или Enhydra, которые обеспечивают отделение сеансов и предоставляют службы аутентификации для программ внутри них. Недостаток данных программ равноценен их достоинству: они облегчают сохранение на сервере информации о состоянии по каждому пользовательскому сеансу. Хранение данных сведений может оказаться проблемой. Оно занимает ресурсы, и данную информацию необходимо удалять по истечении определенного времени, поскольку между транзакциями не существует способа точно определить, остается ли пользователь на связи.

Как обычно, наилучшая рекомендация — выбирать простейшую из возможных моделей, т.е. сопротивляться соблазну создавать тяжеловесную конструкцию, опирающуюся на Java или сервер приложений, в ситуациях, когда решить задачу можно с помощью простых CGI-сценариев и cookie-файлов.

Проблемой методики использования браузера в качестве универсального клиента является то, что CGI-серверы очень непросто отделить от среды браузера, поэтому может быть трудно написать сценарий или автоматизировать транзакции к серверу. Unix-ответ — трехуровневая архитектура: Web-формы, вызывающие CGI-сценарии, которые в свою очередь вызывают команды. Интерфейсом автоматизации являются команды.

Способ, с помощью которого браузеры отделяют клиентов от серверов, имеет более серьезные последствия. В Web-среде стало сложнее и не так привлекательно привязывать потребителей к закрытым, частным протоколам и API-интерфейсам. Экономика разработки программ, таким образом, склоняется к HTML, XML и другим открытым, текстовым Internet-стандартам. Данная тенденция успешно и интересно объединяется с развитием модели разработки открытого исходного кода, которая рассматривается в главе 19. В мире, где Web является созидательной средой, Unix-традиции проектирования, включая рассмотренные в настоящей главе методики проектирования интерфейсов, выглядит гораздо более естественно, чем когда-либо прежде.

11.9. Молчание — золото

Рассматривая тему интерактивных пользовательских интерфейсов нельзя не упомянуть одну из наиболее давних и постоянных идей проектирования в Unix — правило тишины. В главе 1 отмечалось, что хорошо спроектированные Unix-программы, не имеющие интересной или неожиданной для пользователя информации, должны выполнять свою работу бесшумно. Для этого существуют веские причины, которые намного "пережили" медленные телетайпы, на которых родилась операционная система Unix.

Одна из них заключается в том, что программы, которые выдают излишнюю информацию, часто не способны надежно взаимодействовать с другими программами. Если CLI-программа отправляет сообщения о состоянии на стандартный вывод, то программы, пытающиеся интерпретировать данный вывод, столкнутся с проблемой обработки или отклонения данных сообщений (даже если ничего неординарного не произошло). Гораздо лучше отправлять только реальные ошибки в стандартный вывод сообщений об ошибках и вообще не выводить незапрашиваемые данные.

Другой проблемой является то, что вертикальное пространство пользовательского экрана весьма ценно. Каждая строка бесполезных данных, отображаемых программой, означает потерю одной строки дисплея.

Третья проблема: бесполезные сообщения отвлекают внимание пользователя. Они являются еще одним источником отвлекающих перемещений по экрану дисплея, который может быть посредником в решении более важных приоритетных задач, таких как общение с другими пользователями.

Необходимо двигаться дальше и снабжать длительные операции индикаторами выполнения. Это хороший стиль, позволяющий пользователю, распределять свое внимание на чтение почты или другие задачи, ожидая подтверждения. Однако не следует загромождать GUI-интерфейсы всплывающими окнами подтверждения, кроме случаев, когда необходимо обезопасить операции, в результате которых данные могут быть утеряны или повреждены, и даже тогда необходимо скрывать их, если родительское окно свернуто, и не показывать до тех пор, пока родительское окно не получит фокус[107]. Задача дизайнера интерфейса заключается в том, чтобы помогать пользователю, а не беспричинно вмешиваться в его работу.

Как правило, считается плохим стилем сообщать пользователю о том, что он уже знает (два классических примера: "Программа <foo> запускается…" или "Программа <foo> завершает свою работу…"). Конструкция интерфейса в целом должна подчиняться правилу наименьшей неожиданности, а содержание сообщений — правилу

наибольшей неожиданности. Сообщения должны уведомлять пользователя только о

явлениях, отклоняющихся от нормы.

Данное правило звучит еще жестче для запросов на подтверждение. Постоянные запросы на подтверждение, когда ответом почти всегда является "Да", вырабатывают у пользователя привычку нажимать кнопку "Да", не принимая во внимание сути запроса, привычку, которая может иметь весьма печальные последствия. Программы должны запрашивать подтверждение только в случае, если имеется веская причина подозревать, что ответом будет "нет, нет и еще раз нет". Запрос на подтверждение, суть которого не является неожиданной, — признак плохого дизайна. Любые запросы на подтверждение вообще могут означать, что единственное, чего действительно не достает интерфейсу, это команда отмены предыдущего действия.

Если подробные сообщения о выполнении операций необходимы в целях отладки, то по умолчанию они должны быть отключены и вызываться только в случае запуска программы с ключом -v (вывод подробных сведений). Перед окончательным выпуском программы необходимо сделать так, чтобы как можно больше обычных сообщений отображались только в случае использования ключа вывода подробных сведений.

12

Оптимизация

Преждевременная оптимизация — корень всех зол. -Ч. Хоар

Данная глава очень короткая, поскольку главное, чему учит опыт Unix относительно оптимизации производительности, — как узнать, когда не следует выполнять оптимизацию. Второстепенный урок заключается в том, что наиболее эффективные тактики оптимизации обычно связаны с мероприятиями, имеющими иные цели, например, обеспечение четкости конструкции.

12.1. Отказ от оптимизации

Наиболее мощная методика оптимизации, входящая в инструментарий любого программиста, заключается в том, чтобы ничего не делать.

Этот весьма выдержанный в духе Дзэн совет верен по нескольким причинам. Одной из них является экспоненциальный эффект закона Мура — самый разумный, дешевый и часто

самый быстрый способ достичь прироста производительности заключается в том, чтобы подождать несколько месяцев, пока целевое аппаратное обеспечение станет более мощным. Учитывая ценовое соотношение между аппаратным обеспечением и временем программиста, почти всегда существуют лучшие варианты использования времени, чем оптимизация уже работающей системы.

Данная точка зрения может быть обоснована математически. Почти никогда не следует выполнять оптимизацию, которая сокращает использование ресурсов просто на постоянную величину. Гораздо разумнее сконцентрировать усилия на случаях, в которых можно сократить

среднее время запуска или использование пространства с O(

n ?) до O(

n) или O(

n log n)[108] или подобным образом понизить порядок. Согласно закону Мура, линейный прирост производительности достаточно быстро уменьшается[109].

Существует другая весьма конструктивная форма отказа от оптимизации — не писать код. Скорость работы программы не может быть уменьшена кодом, которого в ней нет, она может быть уменьшена кодом, который в ней

есть , если он менее эффективен, чем мог бы быть. Однако это уже другая проблема.

12.2. Измерения перед оптимизацией

Когда имеются реальные доказательства того, что разрабатываемое приложение работает слишком медленно, тогда (и

только тогда) наступает время обдумать оптимизацию кода. Однако перед этим необходимо сделать нечто большее — провести измерения.

Читателям следует вспомнить шесть правил Роба Пайка, описанных в главе 1. Один из первых уроков, который усвоили программисты Unix, заключается в том, что интуиция — неверный советчик при определении местоположения "бутылочных горлышек", причем даже для того, кто очень хорошо знает свой код. Unix-системы, в отличие от большинства других операционных систем, обычно поставляются вместе с профайлерами (profiler — подпрограмма протоколирования), и их стоит использовать.

Чтение результатов работы профайлера вполне можно сравнить с искусством. Существует несколько периодически повторяющихся проблем: первая — погрешность измерения, вторая — влияние налагаемых внешних задержек, третья — перевес верхних узлов графа вызовов.

Фундаментальной является проблема погрешности измерений. Профайлеры выполняют свои функции, внедряя инструкции, которые фиксируют время выполнения в точках входа и выхода из подпрограмм, а также в фиксированных интервалах во внутреннем коде программ. Выполнение данных инструкций само по себе отнимает некоторое время. В результате сокращается разброс времени вызовов: очень короткие подпрограммы часто выглядят более дорогостоящими, чем они есть на самом деле, с большим количеством шума в их относительном времени вызова, тогда как для более длинных подпрограмм издержки измерения незаметны.

Принимая во внимание погрешность измерения, разумно предположить, что значения времени, указанные для самых быстрых, кратчайших подпрограмм, из-за шума будут завышены. Их выполнение также потребует большого количества времени, если они будут вызываться очень часто, однако, следует уделить должное внимание статистическим данным по количеству их вызовов.

Проблема внешней задержки также является фундаментальной. Существуют различные виды задержек и искажений, которые могут возникать вне поля зрения профайлера. Простейшим примером являются издержки операций с непредсказуемой задержкой: доступ к дискам и сетям, заполнение кэша, переключение контекста процессов и им подобные.

Проблема заключается не столько в возникновении данных задержек — возможно, именно их и требуется измерить, особенно если основное внимание уделяется производительности системы в целом, а не просто регулировке критически важного внутреннего цикла. Реальная проблема состоит в том, что они содержат случайный компонент, а это означает, что результаты любого отдельного запуска профайлера могут быть не самыми достоверными.

Одним из способов минимизации влияния указанных источников шума и получения более четкой картины утечки времени в среднем случае является сложение результатов от множества запусков профайлера. Существует множество весомых причин для создания средств тестирования и тестовых нагрузок до оптимизации разрабатываемых программ. Наиболее важной причиной, как правило, гораздо более важной, чем регулировка производительности, является то, что впоследствии, по мере модификации программы, можно использовать возвратное тестирование ее корректности. После того как это сделано, возможность выполнять профилирование повторяющихся тестов под нагрузкой является хорошим побочным эффектом, который часто предоставляет более точную информацию, чем несколько запусков вручную.

Различные факторы склонны перекладывать затраты времени на вызывающие программы, а не на вызываемые, что увеличивает вес верхних узлов графа вызовов. Например, издержки вызова функции часто относят к вызывающей программе (так это или нет, в некоторой степени зависит от архитектуры конкретной машины и от того, где профайлеру разрешено размещать пробы). Макросы и встраиваемые функции, в случае если компилятор их поддерживает, не показываются в отчете профайлера вообще. Расходуемое ими время относится к вызывающей функции.

Более важно то, что многие средства учета времени создают такое впечатление, будто время, затраченное на подпрограммы, относится к вызывающей программе. (Данная особенность характерна для профайлера

gprof(1) , распространяемого в составе Unix-систем с открытыми исходными кодами). Простое вычитание времени вызываемой программы из времени вызывающей не даст достоверных результатов, если одну и ту же программу могут вызывать несколько других программ — результатом было бы искусственное сокращение времени всех вызывающих программ. Особенно опасным является распространенный случай функции с несколькими узлами вызова, одни из которых создают множество простейших вызовов, а другие создают несколько сложных.

Для получения более прозрачных результатов следует организовать код так, чтобы программы верхнего уровня содержали как можно больше обращений к программам более низкого уровня, а не ко встроенному коду. Если издержки управляющей логики верхнего уровня остаются минимальными, то структура вызова кода будет стремиться организовать отчет профайлера таким способом, который будет сравнительно простым для понимания.

Использование профайлеров еще больше проясняет ситуацию, если в меньшей степени рассматривать их как способы накопления отдельных показателей производительности, и в большей степени как способы определения закономерности, по которой производительность изменяется как функция от интересующих параметров. Такими параметрами могут быть, например, размер проблемной области, частота процессора, скорость диска, размер памяти, оптимизация компилятора или другие релевантные факторы. Необходимо попытаться подобрать модель для данных чисел, используя программное обеспечение с открытым исходным кодом, такое как R, или качественный коммерческий инструмент, подобный MATHLAB.

Естественное сглаживание данных, определяемое моделью, характеризуется выявлением главных факторов и пренебрежением второстепенных, связанных с шумом. Например, при

использовании кубической модели в подпрограмме обращения матрицы в MATHLAB на случайных матрицах от 10x10 до 1000x1000, очевидно, что фактически получается 3 куба с четко определенными границами, которые примерно соответствуют областям "в кэше", "в памяти, но вне кэша" и "вне памяти". Данные демонстрируют такой эффект, даже если не искать его, а просто изучать отклонения от наилучших результатов. Стив Джонсон.

12.3. Размер кода

Наиболее эффективный способ оптимизировать код заключается в том, чтобы сохранять его небольшой размер и простоту. Ранее в данной книге уже рассматривалось множество весомых причин для сохранения небольшого размера и простоты кода. В данной главе рассматривается еще одна такая причина: необходимо, чтобы центральные структуры данных и циклы в коде, время выполнения которых критически важно, никогда не выходили за пределы кэша.

Рассмотрим целевую машину как иерархию типов памяти, упорядоченных по удаленности от процессора. Она включает в себя собственные регистры процессора; его конвейер инструкций; кэш первого уровня (L1); кэш второго уровня (L2); вероятно, кэш третьего уровня (L3); оперативная память (которая среди специалистов старой школы Unix до сих пор изящно называется основой (core)); и дисковые накопители, на которых располагается область подкачки. Такие технологии, как SMP, кластеры с общей памятью и технология доступа к неоднородной памяти (Nonuniform Memory Access — NUMA) добавляют больше уровней в картину, но только расширяют общий разброс.

Любые виды доступа к данному стеку ускоряются. Циклы процессора являются почти бесплатными, исключая несколько требовательных приложений, таких как моделирование ядерных взрывов или сжатие видео в реальном времени. Однако также по мере возрастания скорости процессора, происходит увеличение соотношения скоростей между уровнями в иерархии хранения. Таким образом, относительная стоимость потерь кэша увеличивается.

Наблюдается интересный парадокс. По мере того как стоимость аппаратных ресурсов резко снижается, ожидаемая стоимость крупных структур данных падает, однако, поскольку разница стоимости между смежными уровнями кэша растет, величина производительности, необходимая для выхода за пределы кэша, также возрастает.

"Малое прекрасно" — эта идея, следовательно, является более убедительной, чем когда-либо, особенно в отношении центральных структур данных, которые должны располагаться в как можно более быстром кэше. Данная рекомендация также применима и к коду; средняя инструкция затрачивает больше времени при загрузке, чем при выполнении.

Это меняет некоторые традиционные советы на прямо противоположные. Оптимизация компилятора, подобная развертке цикла, которая освобождает сравнительно дорогие машинные инструкции в обмен на увеличение общего размера кода, может оказаться более нецелесообразной. Другим примером является предвычисление небольших таблиц — например, таблица значений функции sin(x) от величины угла для оптимизации вращения в ядре 3D графики потребует на современной машине 365x4 байт. До того как процессоры стали быстрее, чем память, чтобы требовать кэширования, это было очевидной оптимизацией скорости. В настоящее время, возможно, быстрее будет пересчитывать результаты каждый раз, чем расплачиваться за дополнительные потери кэша, вызванные хранением таблицы.

Однако в будущем, по мере того как размеры кэша возрастут, все может вернуться на свои

места. В общем случае множество видов оптимизации являются временными и могут привести к прямо противоположным результатам по мере изменения соотношения цен. Единственный путь узнать это заключается в измерении и анализе.

12.4. Пропускная способность и задержка

Другим последствием использования быстрых процессоров является то, что производительность обычно ограничивается затратами на I/O-операции и (особенно в случае программ, использующих Internet) затратами на сетевые транзакции. Следовательно, разработчику полезно знать, как проектировать сетевые протоколы для достижения высокой производительности.

Наиболее важной проблемой является максимальное предотвращение полных циклов протокола. Каждая протокольная транзакция, требующая квитирования, превращает любую задержку в соединении в потенциально серьезное замедление. Избежание квитирования не является специфической и традиционной практикой Unix, однако здесь необходимо упомянуть данный практический прием, поскольку из-за квитирования значительно понижается производительность многих протоколов.

О задержке я могу сказать не много. Версия X11 далеко "оторвалась" от X10 в предотвращении полных циклов обращения: Render-расширение уходит еще дальше. X (и в настоящее время HTTP/1.1) является протоколом потоковой передачи. Например, мой портативный компьютер способен выполнять более 4

млн. прямых запросов (8 млн. холостых запросов) в секунду. Однако полные циклы "запрос-ответ" в сотни или тысячи раз дороже. Каждый раз, когда клиент может выполнить какую-либо операцию, не обращаясь к серверу, является огромным выигрышем.Джим Геттис.

Действительно хорошее практическое правило заключается в том, чтобы проектировать конструкцию с наименьшей возможной задержкой и игнорировать затраты полосы пропускания до тех пор, пока профайлеры не укажут обратное. Проблемы, связанные с полосой пропускания, можно решить позднее при разработке с помощью таких технических приемов, как сжатие данных протокола на лету. Однако освободиться от высокой задержки, встроенной в существующую конструкцию, гораздо труднее (часто практически невозможно).

Несмотря на то, что данный эффект наиболее четко проявляется в конструкции сетевых протоколов, компромисс между пропускной способностью и задержкой является гораздо более общим феноменом. При написании приложений программист иногда сталкивается с необходимостью выбора: однократное выполнение дорогостоящих вычислений в расчете на то, что результаты будут использоваться несколько раз, или выполнение вычислений только в случае действительной необходимости (даже если это означает частое перевычисление результатов). В большинстве подобных случаев правильный подход склоняется в сторону низкой задержки. То есть не следует пытаться выполнить дорогостоящее предвычисление в случае, если нет определенных требований к пропускной способности или если изменения не показывают слишком низкую пропускную способность. Предвычисления могут показаться эффективными, поскольку они минимизируют общее использование процессорных циклов, но процессорные циклы дешевы. Если создается простая программа, а не одно из немногочисленных гигантских приложений с интенсивными вычислениями, например, для анализа больших массивов информации, визуализации анимации или упомянутое выше модулирование взрывов, то обычно наилучший путь — предпочесть небольшое время запуска и быстрый отклик.

В ранние дни Unix данную рекомендацию сочли бы еретической. В то время процессоры были гораздо медленнее, а соотношения затрат сильно отличались. Кроме того, модель использования Unix больше склонялась к серверным операциям. Отчасти отметить значение низкой задержки необходимо потому, что даже более молодые Unix-разработчики иногда наследуют давние культурные предубеждения по поводу оптимизации по пропускной способности. Однако времена изменились.

Разработаны три общие стратегии сокращения задержки: (a) пакетные транзакции, способные распределить начальные затраты, (b) разрешение совмещения транзакций и (c) кэширование.

12.4.1. Пакетные операции

Графические API-интерфейсы часто пишутся в предположении, что фиксированные начальные затраты для обновления физического экрана достаточно высоки. Следовательно, операции записи фактически модифицируют внутренний буфер. Программисту придется решить, когда накоплено достаточное количество обновлений, и использовать вызов, который превратит их в обновление физического экрана. Верный выбор промежутка времени между физическими обновлениями может создавать значительные различия в восприятии графических клиентов. Таким способом организованы как X-сервер, так и библиотека

curses(3) , используемая в rogue-подобных программах.

Постоянные служебные демоны представляют собой более характерный для Unix пример пакетирования. Существует две причины для написания постоянно работающих демонов (в противоположность CLI-серверам, которые запускаются заново для каждого сеанса) — очевидная и неочевидная. Очевидной причиной является управление обновлениями общего ресурса. Менее очевидная причина, которая имеет место даже для демонов, не обрабатывающих обновления, заключается в том, что при многочисленных запросах сокращаются затраты на чтение в базе данных демона. Идеальным примером такого демона является DNS-служба

named(8) , которая нередко должна обрабатывать тысячи запросов в секунду, каждый из которых может фактически блокировать загрузку Web-страницы пользователем. Одна из тактик, делающих

named(8) быстрой службой, заключается в том, что в ней дорогостоящие операции анализа дисковых текстовых файлов, описывающих DNS-зоны, заменяются доступом к кэшу, который содержится в памяти.

12.4.2. Совмещение операций

В главе 5 сравнивались протоколы POP3 и IMAP для опроса удаленных почтовых серверов. При этом было отмечено, что IMAP-запросы (в отличие от POP3-запросов) маркируются идентификатором запроса, сгенерированным клиентом. Сервер, отправляя обратно ответ, включает в него метку запроса, к которому относится данный ответ.

POP3-запросы должны обрабатываться клиентом и сервером в строгом порядке. Клиент отправляет запрос, ожидает ответ на данный запрос и только после его получения может

подготовить и отправить следующий запрос. IMAP-запросы, с другой стороны, маркируются, поэтому их передачу можно совместить. Если IMAP-клиенту известно, что требуется доставить несколько сообщений, то он может отправить IMAP-серверу поток из нескольких запросов на доставку (каждый со своей меткой), не ожидая ответов между ними. Маркированные ответы отправятся обратно, как только сервер будет готов. Ответы на более ранние запросы могут поступить в то время, когда клиент все еще отправляет более поздние запросы.

Описанная стратегия является распространенной не только в области сетевых протоколов. Когда требуется сократить задержку, блокировка или ожидание немедленных результатов — крайне неэффективные методы.

12.4.3. Кэширование результатов операций

Иногда можно получить оба преимущества (низкую задержку и хорошую пропускную способность) путем вычисления дорогостоящих результатов по мере необходимости и их кэширования для последующего использования. Выше было сказано, что в службе

named задержка сокращается путем пакетирования. Кроме того, задержка в

named уменьшается с помощью кэширования результатов предыдущих транзакций с другими DNS-серверами.

Кэширование имеет собственные проблемы и компромиссные решения, которые хорошо иллюстрируются одним примером: использование двоичного кэша для устранения издержек синтаксического анализа, связанного с файлами текстовых баз данных. В некоторых вариантах операционной системы Unix данная методика использовалась для ускорения доступа к парольной информации (обычным обоснованием было сокращение задержки при регистрации в системе на очень крупных узлах).

Для того чтобы обеспечить соответствующую отдачу от использования данной методики, необходимо чтобы весь код, обращающийся к бинарному кэшу, проверял временные метки на обоих файлах и обновлял кэш в случае, если мастер-текст имеет более позднюю метку. С другой стороны, все изменения мастер-текста должны выполнятся посредством упаковщика, который обновляет двоичный формат.

Несмотря на то, что данный подход оказывается работоспособным, он обладает всеми недостатками, которые можно ожидать при нарушении правила SPOT. Дублирование данных означает, что в результате не будет никакой экономии пространства, т.е. это исключительно оптимизация скорости. Однако реальная проблема данного подхода заключается в том, что код для обеспечения когерентности между кэшем и мастер-текстом печально известен как "текучий" и чреватый ошибками. Очень частое обновление файлов кэша может привести к неочевидной конкуренции просто ввиду односекундного разрешения временных меток.

Но когерентность все-таки можно гарантировать в простых случаях. Одним из них является интерпретатор языка Python, который компилирует и сохраняет на диске файл p-кода с расширением .pyc, когда файл библиотеки Python импортируется впервые. При последующих проходах кэшированная копия p-кода загружается, в случае если источник с тех пор не изменялся (это позволяет избежать повторного синтаксического анализа исходного кода библиотеки при каждом проходе). Подобная методика используется в Emacs Lisp (файлы .el и . .elc). Данная методика работоспособна, так как доступ для чтения и для записи кэша осуществляется посредством одной программы.

Однако, когда модель обновления главных файлов более сложна, код обеспечения синхронизации склонен к утечкам. Unix-варианты, использующие данную методику для ускорения доступа к критически важным системным базам данных, имели дурную репутацию среди системных администраторов.

Как правило, файлы бинарного кэша являются хрупкой методикой, и, вероятно, будет лучше избегать их использования. Усилия, затраченные на реализацию специализированных средств для сокращения задержки, в данном случае было бы целесообразнее направить на совершенствование конструкции приложения, так чтобы в ней не было "бутылочного горлышка", или даже на тонкую настройку для повышения скорости файловой системы или реализации виртуальной памяти.

Если ситуация на первый взгляд требует использования методики кэширования, то разумно будет взглянуть на уровень глубже и ответить на вопрос: почему вообще понадобилось кэширование? Решение данной проблемы вполне может оказаться не сложнее, чем правильное решение всех граничных случаев в кэширующем программном обеспечении.

13

Сложность: просто, как только возможно, но не проще

Все следует делать так просто, как только возможно, но не проще. —Альберт Эйнштейн

В конце главы 1 философия Unix была сведена к общему принципу — K.I.S.S. (Keep It Simple, Stupid! Будь проще!). В части "Проектирование" данной книги одной из ключевых тем была важность сохранения максимально возможной простоты конструкции и реализации. Однако,

что значит "просто, как только возможно"?

Рассмотрение данного вопроса откладывалось до настоящей главы потому, что простота — комплексное понятие. В качестве теоретической основы при изучении данной темы необходимы некоторые идеи, которые были сформулированы ранее в части "Проектирование", особенно в главах 4 и 11.

Важными вопросами в данной главе являются основные предубеждения Unix-традиции. Некоторые из этих предубеждений стали причинами идеологических войн, продолжавшихся в течение десятилетий. Данная глава начинается с рассмотрения установившейся практики и терминологии Unix, а затем несколько выходит за рамки этой тематики. Автор не пытается сформулировать здесь простые ответы на данные вопросы, это невозможно, однако концептуальные средства, описанные в настоящей главе, вероятно, позволят читателям продвинуться дальше в собственном поиске ответов.

13.1. Сложность

Как и в случае рассмотренных выше вопросов модульности и проектирования интерфейсов, Unix-программисты воспринимают ряд отличий, которые они часто усваивают из опыта, даже не зная, как их назвать. Поэтому начинать следует с изложения некоторых терминов.

Раздел начинается с определения того, в чем заключается сложность программного обеспечения. Далее определяются некоторые горизонтальные границы между несколькими разновидностями сложности. Завершается данный раздел определением еще более важных вертикальных различий между видами сложности, с которыми приходится смириться, и видами сложности, которые можно устранить.

13.1.1. Три источника сложности

Вопросы о простоте, сложности и верном размере программного обеспечения вызывают бурные споры в Unix-сообществе. Unix-программисты развили такое мировоззрение, согласно которому простота — это красота, изящество и добро, а сложность — уродство, абсурдность и зло

В основе программистского стремления отстаивать простоту лежит прагматичный факт: сложность дорого обходится. Сложное программное обеспечение труднее анализировать, тестировать, отлаживать, сопровождать, но прежде всего его сложнее изучать и использовать. Издержки сложности, не устраненные во время разработки, резко проявляются после внедрения программы. Сложность создает источники ошибок, из которых они распространяются и создают проблемы на протяжении всего срока службы программного обеспечения.

Тем не менее, остается достаточно факторов, которые "толкают" программистов "в трясину сложности". В предыдущих главах рассматривалось множество неконтролируемых факторов. Два наиболее широко известных — сползание функций и преждевременная оптимизация. Традиционно Unix-программисты отклоняют эти тенденции, провозглашая с религиозной страстью лозунги, осуждающие всякую сложность.

Что конкретно автор называет "сложностью"? Этот момент достоин точного определения ввиду своей важности.

Unix-программисты (как и остальные программисты) склонны фокусировать внимание на

сложности реализации — по существу, степени трудности, с которой столкнется программист, пытаясь понять программу, для того чтобы создать ее ментальную модель или отладить.

С другой стороны, потребители и пользователи склонны видеть сложность в самих понятиях

сложности интерфейса программы. В главе 11 рассматривалась простота и качество, противоположное ей, т.е. мнемоническая нагрузка. Для пользователя сложность тесно связана с мнемонической нагрузкой. Слабая выразительность и лаконичность также имеют значение, в случае если неразвитый интерфейс вынуждает пользователя выполнять множество чреватых ошибками или просто утомительных низкоуровневых операций вместо нескольких высокоуровневых.

Оба описанных выше вида сложности управляют третьим, более простым: общее количество строк кода в системе, т.е.

размер кодовой базы. В понятиях стоимости на протяжении жизненного цикла данный критерий обычно является наиболее важным. Причины, вероятно, связаны с самым важным эмпирическим выводом программной инженерии, который цитировался ранее: плотность дефектов кода, выраженная в количестве ошибок на сотню строк, стремится к постоянному значению независимо от языка реализации. Большее количество строк кода означает

большее количество ошибок, а отладка является наиболее дорогостоящей и трудоемкой частью процесса разработки.

Размер кодовой базы, сложность интерфейса и реализации могут возрастать одновременно, что является обычным результатом сползания функций и причиной особенных опасений программистов. Преждевременная оптимизация не склонна увеличивать сложность интерфейса, но оказывает негативное влияние (часто крайне негативное) на сложность реализации и размер кодовой базы. Однако эти доводы против сложности сравнительно просты. Более серьезные трудности начинаются, когда приходится искать компромисс между тремя данными критериями.

Ранее уже упоминалась одна ситуация, при которой два критерия изменяются в противоположных направлениях: пользовательский интерфейс, который разрабатывался в основном для сохранения простоты реализации или небольшого размера кодовой базы, может просто перекладывать низкоуровневые задачи на пользователя. (В качестве грубого примера, едва ли вообразимого Unix-программистом, но слишком распространенного в других средах, можно представить редактор, не имеющий функции глобальной замены.) Хотя такой вид неудачной конструкции является чрезвычайно распространенным, он не имеет традиционного названия. В данной книге он называется

ловушкой ручного труда .

Стремление сохранить небольшой размер кодовой базы с помощью чрезвычайно плотных и сложных методик реализации может вызвать нарастание сложности реализации в системе, ведущее к неразберихе, которую невозможно исправить. Ранее такое часто случалось, когда для того чтобы втиснуть программу в очень малую систему, приходилось программировать на ассемблере или использовать такие нетривиальные технические приемы, как самомодифицирующийся код. В наши дни такая проблема не распространена, кроме случаев со встроенными системами, но и в них она быстро становится редкой. Данный вид неудачной конструкции не имеет традиционного названия, однако эту проблему можно назвать

ловушкой уплотнения .

Ловушка уплотнения не рассматривается в учебных примерах данной книги, она определена для контраста с противоположной ей проблемой. Вполне возможны ситуации, когда разработчики проекта будут настолько осторожны в вопросе сложности реализации, что постараются отказаться от сложного, но единообразного пути для разрешения целого класса проблем в пользу большого количества дублирующегося специального кода, последовательно решающего каждую задачу. В результате такого подхода увеличивается размер кодовой базы и появляются проблемы сопровождения, более сложные, чем в случае использования унифицированного метода. Например, Web-проект, где для наполнения страниц действительно необходима централизованная реляционная база данных, может вместо этого содержать несколько различных индексированных файлов данных, содержащих информацию, которую необходимо интегрировать заново при каждом создании страницы. Данный вид неудачной конструкции является слишком распространенным. Он не имеет традиционного названия, однако его можно назвать

ловушкой специального кода .

Выше описаны три вида сложности, а также некоторые ловушки, в которые попадают разработчики, пытающиеся избежать сложности[110]. Примеры рассматриваются далее в настоящей главе.

13.1.2. Компромиссы между сложностью интерфейса и реализации

Одно из наиболее ярких замечаний о Unix-традиции, сделанных когда-либо сторонним наблюдателем, содержится в статье Ричарда Гэбриэла (Richard Gabriel), которая называется

"Lisp: Good News, Bad News, and How to Win Big" [25]. Гэбриэл в течение многих лет является бессменным лидером Lisp-сообщества, и данная статья главным образом была доводом в пользу особого стиля Lisp-конструкции. Однако сам автор статьи признает, что сейчас она вспоминается в основном из-за раздела, который называется "

The Rise of Worse Is Better" .

В статье утверждалось, что Unix и C обладают характеристиками вирусов и что в эволюционной борьбе конструкций программного обеспечения такие особенности, как простота реализации и переносимость, которые ведут к быстрому распространению (заражаемости), являются более эффективными, чем корректность и завершенность конструкции. Гэбриэл в своем предвидении так близко подошел к влиянию экспертной оценки на программное обеспечение с открытым исходным кодом, что сообщество открытого исходного кода приняло его как одного из своих теоретиков.

Не так часто вспоминается то, что главный аргумент Гэбриэла относился к весьма специфическому компромиссу между сложностью реализации и сложностью интерфейса, который довольно точно вписывается в рассмотренные выше категории. Гэбриэл сопоставляет философию MIT, в которой наиболее ценится простота интерфейса, с философией Нью-Джерси, где выше всего ценится простота реализации. Затем он утверждает, что хотя MIT-философия приводит к появлению программного обеспечения, которое абстрактно лучше, модель Нью-Джерси (худшая) обладает лучшими характеристиками распространения. Со временем люди уделяют больше внимания программному обеспечению, написанному в стиле Нью-Джерси, поэтому оно быстрее улучшается, и худшее становится лучшим.

В действительности, философия MIT и Нью-Джерси имеет аналоги как конфликтующие направления в самой Unix-традиции проектирования. Один тип Unix-мышления придает особое значение небольшим точным инструментам, конструкциям, разработанным с нуля, а также простым и последовательным интерфейсам. Дуг Макилрой решительно отстаивает данную точку зрения. Другой тип мышления акцентирует внимание на простых реализациях, которые работают и быстро распространяются, даже если используются методы грубой силы, а некоторые граничные случаи необходимо устранить. Код Кена Томпсона, а также его принципы программирования часто и очевидно склоняются к этому направлению.

Конфликт между данными подходами возникает именно потому, что иногда программист может получить простой интерфейс, если согласен заплатить за него сложностью реализации, и наоборот. Оригинальный пример Гэбриэла о том, как системные вызовы, выполняющие длительные операции, обрабатывают прерывания, которые не способны удержать или замаскировать, до сих пор остается одним из лучших. В MIT-философии правильный путь заключается в выходе из системного вызова и автоматическом его возобновлении, как только прерывание будет обработано. Данную методику труднее реализовать, однако она ведет к созданию более простого интерфейса. В философии Нью-Джерси системный вызов возвращает ошибку, указывающую на то, что вызов был прерван и пользователь должен перезапустить его. Такой подход значительно проще в реализации, но приводит к программному интерфейсу, который труднее использовать.

На практике применяются оба подхода. Unix-разработчики старой школы немедленно подумают о сравнении System V и BSD-стилей обработки программных сигналов. Последний следует MIT-философии, тогда как System V-стиль происходит из Нью-Джерси. В основе

выбора между ними лежит сложный вопрос, который непосредственно не касается распространения программного обеспечения: если цель заключается в ограничении глобальной сложности, что разработчик охотнее всего принесет в жертву? Что ему

следует принести в жертву?

Одним эпохальным примером, не упомянутым Гэбриэлом, являются распределенные гипертекстовые системы. Ранние проекты распределенных гипертекстовых систем, такие как NLS и Xanadu, были жестко ограничены предположениями MIT-философии о том, что ссылки, указывающие на несуществующий объект, являются недопустимой неполадкой в пользовательском интерфейсе. Это ограничивало системы либо просмотром только контролируемого, закрытого набора документов (как, например, на одном CD-ROM), либо реализацией различного тиражирования со все возрастающей сложностью, кэширующих и индексирующих методов в попытке предотвратить случайное исчезновение документов. Тим Бернерс-Ли (Tim Berners-Lee) разрубил этот гордиев узел, разрешив проблему в классическом стиле Нью-Джерси. Простота реализации, достигнутая им благодаря разрешению ответа "404: Not Found", была тем, что сделало систему World Wide Web достаточно легковесной, чтобы она могла успешно распространяться.

Сам Гэбриэл придерживался мнения о том, что "худшее" более заразительно и, в конце концов, стремится к победе. Несмотря на это, он неоднократно публично менял свое мнение относительно основополагающего вопроса, связанного со сложностью: сложность — это хорошо или плохо? Его непостоянство отражает множество продолжающихся споров в Unix-сообществе.

Автору данной книги не представляется возможным сформулировать универсальный ответ. Как и большинство других важнейших вопросов в данной главе, хороший вкус и инженерное мышление предполагают различные ответы в разных ситуациях. Важно развить привычку тщательно обдумывать данную проблему для всех и для каждой разрабатываемой конструкции. Как отмечалось ранее при обсуждении модульности программного обеспечения, сложность сопряжена с затратами, которые необходимо весьма тщательно предусматривать.

13.1.3. Необходимая, необязательная и случайная сложность

В идеальном мире Unix-программисты создавали бы только небольшие, совершенные "жемчужины программирования", каждая из которых была бы минимальной, изящной и безупречной. Однако одной из негативных черт реальности является то, что она часто ставит сложные проблемы, требующие сложных решений. Невозможно управлять реактивным лайнером с помощью изящной процедуры из десяти строк кода. Существует слишком много блоков оборудования, множество каналов и интерфейсов, множество различных процессоров, т.е. слишком много различных подсистем, определенных независимыми разработчиками, которые часто не согласны друг с другом даже в фундаментальных вопросах. Даже если предположить, что разработчик добился успеха в изящной реализации всех отдельных частей программного обеспечения для авиационной электронной системы управления, в ходе их интеграции, вероятно, будет создан большой, сложный и нечеткий код с одним достоинством — он действительно будет

работать .

Реактивные самолеты обладают

необходимой сложностью. Существует довольно четкая грань, за которой невозможно

принести в жертву функции в обмен на простоту, поскольку самолет должен оставаться в воздухе. Благодаря самому этому факту, разработчики авиационных электронных систем управления не склонны втягиваться в "религиозные войны" в вопросе сложности, Unix-программисты также часто избегают их.

Конечно, реактивные авиалайнеры не застрахованы от системных сбоев, возникающих из-за чрезмерной сложности. Но вопросы проектирования проще распознать и проанализировать в программном обеспечении, для которого требования более гибкие, а поиски компромисса между предполагаемыми функциями и сложностью просты. (Здесь и далее в настоящей главе понятие "функции" используется в весьма широком смысле, который включает в себя различные элементы, такие как прирост производительности или общая степень изысканности интерфейса.)

Для того чтобы добиться более яркого видения проблемы, необходимо начать с определения отличия между

случайной сложностью и

необязательной сложностью [111]. Случайная сложность возникает вследствие того, что разработчик не нашел простейшего способа реализации заданного набора функций. Такая сложность преодолевается благодаря хорошему проектированию или хорошему перепроектированию. С другой стороны, необязательная сложность связана с некоторой желаемой функцией. Необязательная сложность может быть устранена только путем изменения целей проекта.

Когда разработчики не могут отличить необязательную сложность от случайной, споры по поводу проекта создают тупиковую ситуацию. Вопросы о том, каковы цели проекта, смешиваются с вопросами об эстетике простоты и о том, кто умнее среди участников спора.

13.1.4. Диаграмма видов сложности

Выше были показаны две различные шкалы для анализа сложности. Данные шкалы фактически перпендикулярны друг другу. Рис. 13.1 может помочь при выяснении связей. В каждом из девяти блоков на рисунке приведен общий источник определенного вида сложности.

Рис. 13.1. Источники и виды сложности

Некоторые из данных разновидностей сложности уже рассматривались ранее в данной книге, особенно случайная сложность. В главе 4 было показано, что случайная сложность интерфейса часто обусловлена неортогональностью в конструкции интерфейса, т.е. невозможностью тщательной организации интерфейсных операций, так чтобы каждая из них выполняла только одну функцию. Случайная сложность кода (создание более сложного кода, чем это требуется для решения задачи) часто является результатом преждевременной оптимизации. Случайное увеличение кодовой базы часто является результатом нарушения правила SPOT, дублирования кода или его неудачной организации, в которой трудно распознать возможности для повторного использования.

Необходимую сложность интерфейса обычно невозможно уменьшить без сокращения основных функциональных требований к программному обеспечению (данная тема развивается далее в настоящей главе при изучении учебных примеров). Необходимый размер кодовой базы связан с выбором средств разработки, поскольку, если список функций

сохраняется постоянным, то наиболее значимым фактором в размере кодовой базы, вероятно, является выбор языка реализации (что следует из главы 8).

Источники необязательной сложности наиболее трудно поддаются наглядному обобщению, поскольку они часто зависят от тонких суждений о том, ради каких функций стоит повышать сложность. Необязательная сложность интерфейса часто связана с добавлением удобных функций, которые облегчают работу пользователей, но не являются существенными для работы программы. Необязательное увеличение размеров кода (в предположении, что список доступных пользователю функций и используемых алгоритмов является постоянным) часто может быть следствием различных практических приемов, направленных на то, чтобы сделать код более сопровождаемым. В число таких приемов входит добавление развернутых комментариев, использование длинных имен переменных и т.д. На необязательную сложность реализации часто влияет

все, что касается проекта.

С источниками сложности необходимо бороться различными способами. Размер кодовой базы можно уменьшить с помощью лучших инструментальных средств. Сложность реализации можно уменьшить с помощью более тщательного выбора алгоритмов. Сложность интерфейса необходимо исправлять, тщательнее проектируя взаимодействие программы с пользователем, данный навык предполагает анализ эргономических характеристик и психологии пользователя. Это умение менее широко распространено (и возможно, является более сложным в освоении), чем навык написания кода.

С другой стороны, бороться со сложностью необходимо больше при помощи понимания, чем с помощью различных методов. Разработчик уменьшает случайную сложность, понимая, что существует более простой способ решения задачи. Необязательную сложность можно уменьшить, предметно рассуждая о том, какие функции стоит реализовать. Что же касается необходимой сложности, то ее можно уменьшить только с просветлением, фундаментально переопределяющим решаемую проблему.

13.1.5. Когда простоты не достаточно

Неудачное решение вопроса простоты Unix-программистами заключается в том, что они часто действуют так, будто вся необязательная сложность является случайной. Более того, традиция Unix сильно склоняется к тому, чтобы удалять функции во избежание необязательной сложности.

Тем более, что отстаивать эту позицию не сложно (поистине большая часть данной книги направлена на это). Чистый минимализм позволяет нам порой чувствовать себя виртуозами, а минималистское проектирование — весомое противостояние естественной тенденции программных систем развивать все более сложное сопровождение слабо продуманных функций. Однако вычислительные ресурсы и время размышления человека подобны богатству, и их гораздо интереснее использовать, а не накапливать. Разработчику необходимо определить, когда минимализм в проектировании становится не ценной формой самодисциплины, а просто самоистязанием — способом "приобретения добродетелей" в обмен на отказ от реального

использования богатства, необходимого для выполнения работы.

Это сложный вопрос, его просто превратить в аргумент в пользу полного отказа от хорошей проектной дисциплины. Профессионалы старой школы Unix часто избегают его, опасаясь, что

неспособность как можно жестче противостоять сложности и раздуванию кода неминуемо приводит к порицанию. Однако данный вопрос также является

необходимым. Он будет рассматриваться непосредственно при анализе учебных примеров данной главы.

13.2. Редакторы

Ниже в качестве учебных примеров рассматриваются пять различных Unix-редакторов. При их изучении полезно учитывать набор эталонных задач, перечень которых приводится ниже.

•

Редактирование простого текста. Манипулирование простыми ASCII-файлами (или, принимая во внимание эпоху интернационализации, возможно, Unicode-файлами) без известной редактору структуры выше байтового уровня или, возможно, уровня строки.

•

Редактирование расширенного текстового формата. Редактирование текста с атрибутами, в число которых можно включить изменения шрифта, цвета или другие виды свойств текстовых диапазонов (такие как создание гиперссылки). Редакторы, способные выполнять данные функции, должны иметь возможности преобразования между формой представления атрибутов в пользовательском интерфейсе и формой дискового представления данных (например, HTML, XML или другие расширенные текстовые форматы).

•

Подсветка синтаксиса языков программирования. Редактор с подсветкой синтаксиса "знает" о существовании грамматики входных событий и осуществляет такие функции, как автоматическое изменение уровня отступов, если распознает начало или конец блока в языке программирования. В таких редакторах синтаксис обычно выделяется цветом или различающимися шрифтами.

•

Синтаксический анализ вывода неинтерактивных команд. Наиболее распространенный случай данной функции в мире Unix — запуск C-компиляции из редактора, перехват сообщений об ошибках и возможность последующего пошагового выполнения ошибочных блоков без необходимости выхода из редактора.

•

Взаимодействие с постоянными вспомогательными подпроцессами, которые поддерживают состояние между командами редактора. Если такая возможность предусмотрена, то это дает следующие преимущества.

• Появляется возможность управлять системой контроля версий из редактора, отмечая файлы и возвращая изменения в систему без перехода в оболочку или отдельную утилиту.

• Появляется возможность подключать редактор в качестве пользовательского интерфейса к символическому отладчику, чтобы (например) при остановке выполнения программы в контрольной точке автоматически открывался соответствующий файл и подсвечивалась

строка.

• Появляется возможность редактирования файлов на других машинах, если заставить редактор распознавать ссылки на удаленный узел (распознавание такого синтаксиса как /пользователь@узел:/путь/к_файлу). Если у пользователя редактора имеются соответствующие права доступа, то редактор может автоматически запускать такую утилиту, как

scp(1) или

ftp(1) , для загрузки локальной копии файла, а затем автоматически копировать отредактированную версию обратно на удаленный узел в момент сохранения файла.

Все рассматриваемые ниже редакторы способны редактировать простой текст. (Читателю не следует принимать данную возможность как должное, существует множество программ, именуемых редакторами, такие как "текстовые процессоры", которые слишком специализированы для выполнения данной функции.) Переменные степени необязательной сложности будут очевидны, если проанализировать то, как редакторы выполняют более сложные задачи.

13.2.1.

ed

ed(1) представляет собой действительно минималистский Unix-инструмент для редактирования простого текста. Он датируется временем телетайпов[112]. Редактор имеет простой, "аскетический" CLI-интерфейс без экрана. В приведенном ниже листинге компьютерный вывод выделен

курсивом .

ed sample.txt

sample.txt: No such file or directory

#Это строка комментария, а не команда.

#Выше указанное сообщение предупреждает о том,

#что только что создан новый файл sample.txt.

a

the

quick brown fox

jumped over the lazy dog

.

#Выше была команда присоединения, которая добавляет

#к данному файлу текст.

#Сама по себе точка в строке ограничивает добавление текста,

ls/f[a-z]x/dragon/

#В строке 1 заменить первое вхождение подстроки, соответствующей

#символу f с последующим алфавитным символом в нижнем регистре,

#за которым следует символ x, подстрокой 'dragon'.

#Команда замены допускает использование базовых

#регулярных выражений.

1,$P

the

quick brown dragon

jumped over the lazy dog

#Вывод на экран всех строк с первой до последней.

w

51

#С помощью данной команды файл записывается на диск. Команда

#завершает сеанс редактирования.

q

Читателю это может показаться невероятным, но большая часть первоначального кода операционной системы Unix была написана с помощью данного редактора. Читатель с опытом работы в DOS может узнать в данном случае оригинал, с которого был (грубо) смоделирован редактор

EDLIN .

Если задача редактора определяется, как возможность для пользователя создавать и изменять текстовые файлы, то

ed(1) полностью соответствует данному определению. Многие Unix-программисты старой школы почти серьезно утверждают (а некоторые верят в это вполне серьезно), что все редакторы с большим количеством функций, чем имеет ed, являются просто раздутыми.

Уместно подчеркнуть, что редактор

ed был создан Кеном Томпсоном как продуманное упрощение более раннего редактора

qed [71], который был очень похожим (и был первым редактором, использующим регулярные выражения характерным для Unix способом), но имел возможность работать с несколькими буферами, намеренно отброшенную Кеном Томпсоном. Кен Томпсон решил, что данная функция не стоит дополнительной сложности.

Выдающимся свойством редактора

ed(1) и всех его потомков является объектный формат его команд (в примере сеанса показан явный диапазон в команде "p"). Существует сравнительно мощный синтаксис для определения диапазонов строк либо в числовом виде, либо с помощью соответствующий регулярных выражений, либо по специальным стенографическим символам для текущей и последней строки. Большая часть операций редактора может быть применена к любому диапазону. Данный редактор представляет собой хороший пример ортогональной конструкции.

В настоящее время редактор

ed(1) главным образом используется в качестве программно управляемого инструмента в сценариях. Заметим, что редакторы с более сложными режимами интерактивности для этого непригодны. Существует близкий вариант данного редактора,

ex(1) , в котором добавлено несколько полезных функций интерактивности, таких как приглашение на ввод команды. Он иногда полезен в редких случаях, когда редактирование необходимо осуществлять посредством медленной последовательной линии, или в необычных ситуациях восстановления системы после сбоев, когда библиотека поддержки, необходимая для работы других редакторов, не доступна. По этим причинам в каждой Unix-системе включена реализация редактора

ed , а большинство систем включают в себя также редактор

ex .

Потоковый редактор

sed(1) , упомянутый в главе 9, также близко связан с ed. Многие из основных команд аналогичны, хотя предназначены для вызова с помощью ключей командной строки, а не из стандартного ввода.

Почти все Unix-программисты достаточно отклонились от пути строгих и минималистских достоинств и обычно используют редакторы, которые, как минимум, представляют rogue-подобный экранный интерфейс. Однако тот факт, что культ ed существует, красноречиво говорит о том, что он достоин внимания при изучении Unix-стиля.

13.2.2. vi

Оригинальный редактор

vi(1) был первой попыткой надстроить визуальный, rogue-подобный интерфейс на командный набор

ed(1). Как и в ed, команды в редакторе vi представлены отдельными нажатиями клавиш, и он особенно хорошо подходит для операторов, владеющих машинописью.

В первоначальной версии vi отсутствовала поддержка мыши, меню редактирования, макросов, назначаемых клавиш или любой другой формы пользовательской настройки. Сохраняя приверженность культу ed, сторонники редактора vi считали отсутствие данных функций достоинством. С этой точки зрения одним из наиболее важных преимуществ vi является то, что пользователь на новой Unix-системе может немедленно приступить к редактированию без необходимости носить с собой настройки или беспокоиться о том, что стандартные привязки команд будут серьезно отличаться от привычных.

Правда одна характерная черта редактора vi не нравится начинающим пользователям. Это связано с его краткими одноклавишными командами. Редактор имеет

модальный интерфейс — пользователь либо работает в командном режиме, либо в режиме вставки текста. В режиме вставки текста единственными работающими командами являются нажатия клавиши ESC для выхода из режима и (в более новых версиях) клавиши управления курсором. Ввод текста в командном режиме будет интерпретироваться как команда и приведет к случайным (и, возможно, деструктивным) действиям с содержимым файла.

С другой стороны, поклонники vi особенно восхваляют такое свойство командного набора, как объектный формат операций, унаследованный от ed. Большинство расширенных команд также позволяет естественно оперировать любым строковым диапазоном.

В течение многих лет своего существования редактор vi значительно увеличился в размерах. В современные версии включена поддержка мыши, меню редактирования, возможность отмены неограниченного числа последних операций (оригинальный vi допускал отмену только одной последней операции), возможность редактирования нескольких файлов в отдельных буферах и настройка с помощью конфигурационного файла. Однако использование конфигурационных файлов до сих пор остается необычным и, в противоположность Emacs, в vi никогда не было модным использовать встроенный универсальный язык сценариев. Вместо этого в реализациях vi развились отдельные возможности для выполнения таких операций, как подсветка синтаксиса C-кода и синтаксический анализ сообщений об ошибках C-компилятора путем добавления C-кода непосредственно в vi. Взаимодействие с подпроцессами не поддерживается.

13.2.3.

Sam

Редактор

Sam [113] был написан Робом Пайком в Bell Labs в середине 1980-х годов. Sam был предназначен для операционной системы Plan 9, обзор которой приводится в главе 20. Несмотря на то, что данный редактор не был широко известен за пределами Bell Labs, его предпочитали многие разработчики оригинальной Unix, которые упорно продолжали работать над Plan 9, включая самого Кена Томпсона.

Sam — явный и непосредственный потомок редактора ed и остается гораздо более близким к нему, чем vi. Sam включает в себя только две новые концепции: текстовый дисплей curses-стиля и возможность выделения текста при помощи мыши.

В каждом сеансе Sam имеется ровно одно командное окно и одно или несколько текстовых окон. В текстовых окнах редактируется текст, а командные окна принимают команды редактирования в стиле ed. Мышь используется для перемещения между окнами и для выделения диапазонов текста внутри текстовых окон. Sam представляет собой четкую, ортогональную, немодальную конструкцию, в которой устранена большая часть интерфейсной сложности vi.

Большинство команд по умолчанию оперируют с выбранным диапазоном, который можно выделить с помощью мыши. Выбор диапазона для команды также можно осуществить путем указания строкового диапазона как в ed, однако Sam получил значительное преимущество, благодаря тому факту, что пользователь может выбрать меньший диапазон, чем диапазон

строки. Поскольку с помощью мыши можно осуществлять выбор и быстро перемещать фокус между буферами (включая командный буфер), Sam не нуждается в режиме, эквивалентном стандартному (командному) режиму vi. Сотни расширенных команд vi не нужны, а значит, они исключены. В целом, в редакторе Sam добавлено около десятка команд к примерно семнадцати командам vi, и общее количество команд равно примерно тридцати.

Четыре из новых команд в Sam присоединяются к двум унаследованным от

ed(1) и

vi(1) , как способы применения регулярных выражений в целях выбора обрабатываемых файлов и диапазонов. Эти команды обеспечивают в командном языке ограниченные, но эффективные возможности организации циклов и условных операций. Однако не существует способа для именования или записи в параметрической форме процедур командного языка. Командный язык также не может выполнять интерактивное управление подпроцессом.

Интересной особенностью Sam является то, что он разделен на две части. Серверная часть, которая манипулирует файлами и выполняет поиск, отделена от клиентской части, поддерживающей экранный интерфейс. Данный образец модели "разделения ядра и интерфейса" обладает непосредственным преимуществом — несмотря на то, что программа имеет GUI-интерфейс, она может легко работать на медленном соединении при редактировании файлов на удаленном сервере. Кроме того, клиентская и серверная части могут сравнительно просто перенастраиваться.

Sam, как недавно появившиеся версии vi, обладает функцией бесконечной отмены последних операций. Конструктивно данный редактор не поддерживает ни редактирование текста в расширенном формате, ни синтаксический анализ вывода, ни взаимодействие с подпроцессами.

13.2.4. Emacs

Emacs, без сомнения, самый мощный из существующих редакторов для программистов. Он представляет собой большую, изобилующую различными функциями программу с большой гибкостью и возможностью настройки. Как отмечается в разделе "Emacs Lisp" главы 14, Emacs обладает целым встроенным языком программирования, который может применяться для написания неограниченно мощных функций редактирования.

В отличие от vi, Emacs не имеет режимов интерфейса. Вместо этого команды обычно представлены управляющими символами или предваряются нажатием клавиши ESC. Однако в Emacs возможно связать почти любую клавиатурную последовательность с какой-либо командой, а командами могут быть стандартные или специально написанные Lisp-программы.

Emacs способен редактировать несколько файлов, каждый в отдельном буфере, и поддерживает перемещение текста между буферами. Версии программы, работающие в системе X, обладают собственной поддержкой мыши.

Lisp-программы, связанные с командами Emacs, могут осуществлять производные трансформации текста в буфере. Данная возможность интенсивно используется вместе с другими средствами для определения подсветки синтаксиса и режимов редактирования текста в расширенном формате для десятков различных языков и форматов разметки (начиная с поддержки и выделения цветом C-кода, как в vi, но далеко выходя за эти рамки).

Каждый режим представляет собой просто библиотечный файл или Lisp-код, загружаемый по запросу.

Программы на Emacs Lisp способны также интерактивно управлять произвольными подпроцессами. Некоторые примечательные следствия данной возможности были перечислены ранее, включая способность служить в качестве клиентской части для систем контроля версий, отладчиков и тому подобных программ.

Разработчики Emacs[114] создали программируемый редактор, который может иметь развитые логические возможности, которые можно настраивать в нем для сотен различных специализированных задач редактирования. В результате Emacs поддерживает обработку в одном общем контексте всех текстовых данных — файлы, почта, новости, отладочные символы. Данный редактор может служить в качестве настраиваемой клиентской части для любой команды, имеющей интерактивный текстовый интерфейс.

Среди сторонников и противников Emacs бытует анекдот, описывающий его как операционную систему, замаскированную под редактор. Это преувеличение, однако Emacs действительно исполняет роль, отведенную интегрированным средам разработки (Integrated Development Environments — IDE) в Не-Unix-системах (рассмотрение данной темы продолжено в главе 15).

Но перечисленные преимущества сопряжены с некоторыми сложностями. Для того чтобы использовать настроенный Emacs, пользователю необходимо иметь при себе Lisp-файлы, определяющие персональные настройки редактора. Изучение настройки Emacs — целое искусство. Соответственно, Emacs сложнее в изучении, чем редактор vi.

13.2.5. Wily

Редактор

wily [115] — клон редактора

acme [116] из операционной системы Plan 9. Он имеет некоторые общие средства с Sam, но предусмотрен для пользователей с фундаментально отличающимися навыками. Хотя wily, вероятно, распространен менее чем любой другой из рассматриваемых редакторов, он интересен тем, что иллюстрирует отличающийся и, возможно, более близкий к Unix способ реализации Emacs-подобного программируемого редактора.

Редактор wily можно описать как минималистскую IDE-среду, реализацию расширяемости в Emacs-стиле без десятков излишних функций. В Wily даже глобальный поиск и замена, которые являются

обязательным условием для Unix-редакторов, осуществляются внешними программами. Встроенные команды связаны почти исключительно с оконными операциями. В конструкции редактора Wily максимальное использование мыши предусмотрено изначально.

Wily пытается заменить не только традиционные редакторы, но также и традиционные терминальные окна, такие как

xterm(1) . В данном редакторе любой блок текста внутри главного окна (которое содержит множество не перекрывающихся окон Wily) может интерпретироваться как действие или выражение для поиска. Левая кнопка мыши используется для выделения текста, средняя — для выполнения текста как команды (встроенной либо внешней), а правая — для поиска

текста либо в буферах редактора, либо в файловой системе. Никаких постоянных или всплывающих окон не требуется.

В редакторе Wily клавиатура используется

только для ввода текста. Клавиатурные комбинации реализуются не с помощью специального использования клавиатуры, а одновременным нажатием нескольких клавиш мыши. Данные клавиатурные комбинации всегда эквивалентны использованию средней кнопки для какой-либо встроенной команды.

Wily также может применяться в качестве клиентской части для C-, Python- или Perl-программ, сообщая им всякий раз при изменении окна, выполнении команды или поиска с помощью мыши. Данные подключаемые подпрограммы функционируют аналогично режимам Emacs, но не выполняются в адресном пространстве Wily. Вместо этого они сообщаются с редактором посредством очень простого набора удаленных вызовов процедур. Wily поставляется в одном пакете с аналогом

xterm и средством обработки почты, которое использует его как пользовательский интерфейс для редактирования текста.

Так как Wily серьезно зависит от мыши, он не может применяться ни в консольных символьных дисплеях, ни на удаленном канале без передачи X-данных. Как редактор, Wily предназначен для редактирования простого текста. В нем имеется только два шрифта (один пропорциональный и один моноширинный) и отсутствует механизм для поддержки редактирования расширенного текстового формата или подсветки синтаксиса.

13.3. Необходимый и достаточный размер редактора

Ниже приведен анализ учебных примеров с использованием категорий сложности, сформулированных в начале данной главы.

13.3.1. Идентификация проблем сложности

В каждом текстовом редакторе присутствует определенный уровень необходимой сложности. Как минимум, редактор должен поддерживать во внутреннем буфере копию файла или файлов, редактируемых пользователем в текущий момент. Минимальным требованием являются функции для импорта и экспорта файлов данных (обычно с диска и на диск, хотя потоковый редактор

sed(1) является примечательным исключением). Необходима поддержка какого-либо способа модификации буфера, однако этот способ невозможно определить без описания специфических функций, которые являются необязательными. За рамками описанных функций четыре рассмотренных выше примера демонстрируют широкий диапазон отличий в уровнях необязательной и случайной сложности.

Из всех рассмотренных редакторов

ed(1) обладает наименьшей сложностью. Почти единственной неортогональной функцией в его командном наборе является то, что многие его команды могут принимать суффиксы "p"

или "]" для распечатки или вывода результатов выполнения. Даже после трех десятилетий добавления функций существует менее тридцати команд редактирования, а обычный рабочий набор большинства пользователей включает в себя не многим более десятка команд. Не многое можно удалить из редактора в части необязательной сложности, а случайную сложность вообще трудно идентифицировать. Пользовательский интерфейс редактора ed строго компактен.

Оборотной стороной этого является то, что интерфейс редактора ed не вполне подходит даже для таких базовых задач редактирования, как быстрый просмотр текстового файла. Пользователю необходимо довольно жестко ограничить свои цели, для того чтобы ed оказался приемлемым решением для интерактивного редактирования.

Предположим, что в качестве цели добавляется "поддержка визуального просмотра и редактирования множества файлов". В этом случае редактор Sam кажется не очень далеким от минимального ed-расширения, которое могло бы достичь данной цели. Примечателен тот факт, что разработчики не изменили семантики унаследованных ed-команд. Они сохранили существующий, ортогональный набор возможностей и добавили сравнительно небольшое число новых, которые сами по себе являются ортогональными.

Одним крупным увеличением необязательной сложности (сложности реализации) является имеющаяся в редакторе Sam возможность отмены неограниченного числа операций. Другой значительный фактор — новое средство организации циклов и взаимодействия на основе регулярных выражений в командном языке редактора. Данные свойства, а также тот факт, что мышь можно использовать как устройство выбора, — это почти все, что отличает Sam от гипотетического ed с оконным интерфейсом и возможностью использования мыши.

На уровне конструкции в Sam трудно идентифицировать какую-либо случайную сложность, однако в этом нельзя быть уверенным без всестороннего аудита кода. Интерфейс является, как минимум, полукомпактным, а возможно, и строго компактным. Данный редактор функционирует в соответствии с высочайшими стандартами Unix-проектирования, что не удивительно, учитывая его происхождение.

В противоположность Sam, редактор vi выглядит довольно раздутым и недоработанным. В vi имеются сотни команд, многие из которых дублируются, что в лучшем случае является необязательной сложностью, а возможно, и случайной. Предположительно, большинству пользователей известно не более 5 % командного набора. При сравнении данного редактора с примером Sam возникает закономерный вопрос: почему сложность интерфейса vi настолько высока?

В главе 11 описан результат отсутствия стандартных клавиш перемещения курсора в ранних rogue-подобных программах. Редактор vi был одной из таких программ. Когда он появился, его автор знал, что многим пользователям понадобится возможность использовать традиционные для экранных телетайпов Unix клавиши перемещения курсора. Это сделало модальный интерфейс неизбежным. Как только клавиши h, j, k и l получили зависимые от режима значения в буфере редактирования, слишком быстро проявилась привычка добавлять новые команды уникальным способом.

Редактор Sam, с его зависимостью от растрового дисплея с клавишами-стрелками и мышью, может быть гораздо более четким. И является таковым.

Однако нагромождение команд vi является сравнительно поверхностной проблемой. Это интерфейсная сложность, но большинство пользователей могут проигнорировать и игнорируют ее (интерфейс является полукомпактным в том смысле, как это было определено в главе 4). Более глубокой проблемой является ловушка специального кода. За годы своего существования vi постепенно накапливал все больше встроенного узкоспециального C-кода

для выполнения таких задач, которые Sam "отказывается" выполнять, а в редакторе Emacs для этих целей используются модули Lisp-кода и управление подпроцессами. Расширения vi, в отличие от расширений Emacs, не являются библиотеками, загружаемыми по мере необходимости. Пользователи постоянно расплачиваются за распухание кода. В результате различие в размерах между современными версиями редакторов vi и Emacs не такое большое, как можно было бы ожидать. В середине 2003 года на Intel-машине GNU Emacs занимал 1500 Кбайт, тогда как vim занимал 900 Кбайт. Эти 900 Кбайт хранят в себе огромную необязательную и случайную сложность.

Для приверженцев vi отсутствие встроенного языка сценариев, то есть отличие от Emacs, стало вопросом индивидуальности, центральной частью общего мифа о том, что vi является легковесным редактором. Поклонники vi любят говорить о буферах фильтрации с внешними программами и сценариями для выполнения тех задач, которые в Emacs выполняют встроенные сценарии. Однако в реальности команда "!" редактора vi не способна фильтровать меньшие диапазоны буфера редактирования, чем диапазон строк (редакторы Sam и Wily, несмотря на то, что управление подпроцессами в них развито не больше, чем в vi, способны, по крайней мере, фильтровать произвольные текстовые диапазоны, а не только диапазоны строк). Если в vi необходима поддержка файловых форматов и синтаксиса, которые отличаются в более мелких деталях (а в большинстве случаев это так), то сведения об этих форматах и синтаксисе должны быть встроены в C-код. Таким образом, вряд ли соотношение размеров кодовых баз между Emacs и vi будет улучшаться в пользу vi, а, вероятнее всего, оно будет только ухудшаться.

Emacs является достаточно большим редактором с достаточно запутанной историей, что значительно усложняет разделение его необязательной и случайной сложности. Начать можно с попытки отделения незначительных случайных свойств конструкции Emacs от ее необходимых основных элементов.

Возможно, наиболее очевидной незначительной частью конструкции Emacs является язык Emacs Lisp. Lisp существенен для работы редактора, он представляет то, что в настоящее время называется встроенным языком написания сценариев, однако если бы данным языком был Python, Java или Perl, то Emacs немногим отличался бы в своих возможностях. Однако в то время, когда был создан Emacs (в 1970-х годах), Lisp был почти единственным языком, обладающим характеристиками (включая неограниченно расширяемые типы и методику сборки мусора), которые позволяли применять его для данной задачи.

Большинство деталей обработки событий в

emacs и управления растровым дисплеем (включая поддержку интернационализации) также являются случайными. В истории Emacs они привели к "великому расколу" (разветвление GNU Emacs/XEmacs), который демонстрирует тот факт, что ни одна другая часть конструкции не требует какой-либо модели событий.

С другой стороны, способность связывать произвольные последовательности событий с произвольными встроенными или определенными пользователем функциями является необходимой. Язык сценариев может измениться, как может измениться и событийная модель, однако без полиморфизма в способе их соединения конструкция Emacs была бы неузнаваемой и недееспособной. Расширения должны были бы конкурировать друг с другом за монопольное использование ограниченного множества событий, а активизация нескольких взаимодействующих режимов в одном и том же буфере была бы затруднена или невозможна.

Массивная библиотека расширений, поставляемая с Emacs, также является случайной.

Способность создавать такие расширения может быть существенной, но конкретный их

набор, имеющийся в настоящее время, — продукт истории и случая. Все они могут быть изменены или заменены, а в результате остался бы узнаваемый Emacs.

Однако взаимодействие с подпроцессами является необходимым свойством. Без него режимы Emacs были бы не способны реализовать ожидаемую IDE-подобную интеграцию и возможность функционировать в качестве интерфейсной части для многих различных инструментальных средств.

Поучительным является опыт небольших редакторов, клонирующих стандартные привязки клавиш и внешний вид Emacs без эмуляции его расширяемости. Существовало несколько таких клонов, наиболее известными из которых являются, вероятно,

MicroEmacs и pico , однако ни один из них не получил значительного распространения.

Определение случайных и обязательных свойств в конструкции Emacs позволяет понять, какая часть ее сложности является необязательной, а какая случайной. Однако более важно то, что это позволяет увидеть за поверхностными различиями между Emacs и тремя рассмотренными ранее редакторами действительно важное: тот факт, что цели конструкции Emacs являются гораздо более широкими. Emacs претендует на роль унифицированного интерфейса для всех инструментальных средств обработки текста.

Редактор Wily представляет собой интересную противоположность Emacs. Как в случае с Sam, уровень необязательной сложности в Wily низок. Пользовательский интерфейс редактора может быть кратко, но эффективно описан на одной странице.

Однако за это изящество приходится платить. Не существует возможности привязать функции к каким-либо комбинациям клавиш или входным жестам, кроме ограниченного набора аккордов с использованием мыши. Вместо этого каждая функция редактора, кроме самых основных операций вставки и удаления текста, должна быть реализована с помощью внешней программы — в виде отдельного сценария, либо специализированного процесса-симбионта, прослушивающего входные события Wily. (Первая методика основывается на запусках внешней программы, достаточно быстрых, чтобы не вызывать заметной интерфейсной задержки, а это было совсем не так, ни в среде, где появился Emacs, ни в Unix-системах, на которые он был впервые перенесен.)

Необязательная сложность, которая в Emacs реализуется в режимах Lisp-расширений, в Wily распределена по специализированным симбионтам, каждый из которых должен быть совместимым со специальным интерфейсом сообщений данного редактора. Преимущество данного подхода заключается в то, что такие симбионты могут быть написаны на любом выбранном пользователем языке. В дополнение к этому, симбионты (ввиду того, что они запускаются вне редактора) не могут неблагоприятно повлиять друг на друга или на ядро Wily (в отличие от режимов Emacs). Недостатком является то, что сам Wily вообще не способен непосредственно взаимодействовать с обычными инструментальными средствами Unix, вызванными как подпроцессы.

Таким образом, распределенные сценарии

wily не являются таким же мощным средством, как встроенный язык сценариев в Emacs. Соответственно, рамки технических требований к Wily значительно сужены. Создатели редактора, в частности, отказываются от какой бы то ни было поддержки подсвечивания синтаксиса или редактирования расширенного текстового формата, и ни Wily, ни его потомок в Plan 9,

acme, не могут выполнять данные функции.

Все вышесказанное заставляет более четко сформулировать центральный вопрос данной

главы: когда большие цели оправдывают создание большой программы?

13.3.2. Компромиссы не действуют

Сопоставление редакторов Sam и vi явно указывает на то, что, по крайней мере, если рассматривать редакторы, попытки достичь компромисса между минимализмом ed и всеохватывающей полнотой Emacs не имеют значительного успеха. Такие попытки предпринимались разработчиками vi, в результате редактор утратил преимущества минималистской конструкции, но не приобрел достоинств Emacs. Напротив, vi попадает в ловушку специального кода. Этой ловушки избежал редактор Wily, но он не может сравниться с мощностью Emacs и для выполнения серьезных функций требует нестандартного интерфейса сопряжения от каждого из его интерактивных симбионтов.

Очевидно, существует проблема, толкающая редакторы в направлении увеличения сложности. В случае с vi данную проблему идентифицировать не сложно: это стремление к удобству. Несмотря на то, что теоретически адекватным может быть редактор ed, весьма немногие пользователи (кроме, возможно, самого Кена Томпсона) отказались бы от экранного редактирования в стремлении препятствовать раздуванию кода.

В более широком смысле программы-посредники между пользователем и внешним миром особенно богаты функциями. В число таких программ входят не только редакторы, но и Web-браузеры, программы для чтения почты и новостей, а также другие коммуникационные программы. Все они стремятся развиваться в соответствии с законом охвата программ (Law of Software Envelopment), известным также как закон Завински, который гласит: "Каждая программа пытается расширяться до тех пор, пока не сможет читать почту. Программы, которые не могут расширяться до такой степени, заменяются теми, которые могут".

Джейми Завински (Jamie Zawinski), автор этого утверждения (и один из ведущих разработчиков Web-браузеров Netscape и Mozilla), в более широком смысле утверждает, что все действительно полезные программы склонны превращаться в нечто подобное "швейцарскому армейскому ножу". Коммерческий успех крупных, интегрированных пакетов приложений за пределами мира Unix часто подтверждает данную теорию и откровенно ставит под сомнение Unix-философию минимализма.

В пределах своей корректности закон Завински означает, что некоторые программы стремятся быть малыми, а некоторые большими, но средняя масса нестабильна. Поверхностные проблемы vi можно отнести к истории, но более глубокие являются результатом постоянной необходимости добавления функций и одновременного неприятия встроенных сценариев и функций управления подпроцессами, которые у приверженцев vi ассоциируются с чрезмерным размером программы. На другом уровне принятие двух режимов в интерфейсе (командного режима и режима вставки) открыло "банку с червями" — стало гораздо проще добавлять новые команды, не принимая во внимание их влияние на сложность конструкции в целом.

Примеры Emacs и Wily далее подсказывают,

почему некоторые программы стремятся быть большими: для того, чтобы в одном контексте можно было объединить несколько связанных задач. Редактирование и контроль версий (или редактирование и обработка почты, редактирование и отладка и т.д.) являются отдельными задачами с точки зрения разработчиков, однако пользователи часто предпочитают иметь одну большую среду, позволяющую им выделять блоки текста, а не тратить время и отвлекаться, перепрыгивая между несколькими программами, каждой из которых необходимо

передать одно то же имя файла или содержимое некоторого буфера обмена.

В более широком смысле предположим, что рассматривается вся Unix-среда как единое произведение сообщества. Тогда культ "небольших точных инструментов" и стремление сохранять низкий уровень сложности интерфейса и размер кодовой базы могут привести прямо к ловушке ручного труда — пользователю приходится обслуживать весь общий контекст самостоятельно, поскольку инструменты не делают этого.

Возвращаясь к специфике редакторов, Sam показывает, что конструкция vi не верна. Wily — смелая попытка предотвратить крайность Emacs, которая не достигла цели, поскольку Wily не способен поддерживать подсветку синтаксиса. Однако Wily или некоторая реализация конструктивных идей Emacs, очищенных и освобожденных от исторического груза, — возможно, правильное решение. Значение необязательной сложности зависит от целей, определяемых разработчиком, а способность распределения контекста среди всех инструментов, ориентированных на обработку текста и связанных с задачей, является весьма ценной.

13.3.3. Является ли Emacs доводом против Unix-традиции?

Традиционное для Unix видение мира, однако, настолько привязано к минимализму, что в нем не слишком хорошо различаются проблемы специализированного кода vi и необязательная сложность Emacs.

Причиной того, что vi и Emacs никогда не привлекали Unix-программистов старой школы, является то, что эти редакторы

уродливы . Возможно, это обвинение — мнение "старой Unix", но если бы оно не защищало своеобразный стиль старой Unix, то "новой Unix" не существовало бы.Дуг Макилрой.

Нападки пользователей vi на Emacs, наряду с нападками на vi со стороны все еще привязанных к ed радикальных представителей старой школы, — лишь эпизоды в крупном споре, противостоянии изобилия богатства и добродетелей аскетизма. Данный спор связан с конфликтом между стилями Unix старой и новой школ.

"Своеобразный стиль старой Unix" являлся отчасти следствием бедности, точно так же, как японский минимализм, — человек учится делать больше и более эффективно, имея немногое, когда у него нет другого выбора. Однако Emacs (и Unix новой школы, воссозданная на мощных PC-компьютерах и скоростных сетях) — порождение богатства.

Иной была Unix старой школы. В Bell Labs были достаточные ресурсы, поэтому Кен не был ограничен требованиями срочного создания продукта. Вспомним оправдание Паскаля за написание длинного письма ввиду того, что у него не было времени, достаточного для написания короткого. Дуг Макилрой.

С тех пор Unix-программисты поддерживают традицию, в которой изящное превозносится над чрезмерным.

С другой стороны, массивная конструкция Emacs появилась не в Unix, а была создана Ричардом М. Столлменом внутри весьма отличающейся культуры, процветающей в 1970-х годах в лаборатории искусственного интеллекта (Artificial Intelligence Lab) Массачусетского технологического института (MIT). Лаборатория AI MIT была одним из богатейших подразделений академии компьютерных наук Люди учились рассматривать вычислительные

ресурсы как дешевые, предвосхищая позицию, которая в течение последующих пятнадцати лет была бы нежизнеспособной в каком-либо другом месте. Столлмен был равнодушен к минимализму, он искал максимальную мощность и простор для своего кода.

Центральный конфликт в Unix-традиции всегда был и остается между двумя подходами — делать больше с меньшими ресурсами и делать больше с большими. Конфликт снова возникает во множестве различных ситуаций, часто как борьба между конструкциями, имеющими качество четкого минимализма, и конструкциями, в которых выразительность и мощь выбираются даже ценой высокой сложности. Аргументы обеих сторон ("за" или "против" Emacs) иллюстрируют данное напряжение с тех пор, как эта программа в начале 1980-х годов была впервые перенесена на Unix.

Такие же полезные и крупные программы, как Emacs, ставят Unix-программистов в неудобное положение именно потому, что они заставляют сталкиваться с этим конфликтом. Примеры таких программ подсказывают, что минимализм старой школы Unix ценен как наука, но так можно впасть в крайность догматизма.

Существует два пути решения данной проблемы. Один — отрицать, что большая программа действительно является большой. Другой путь состоит в развитии способа анализа сложности, который не является догмой.

Мысленный эксперимент с заменой Lisp и библиотек расширения дает возможность по-новому взглянуть на частое обвинение, что Emacs раздут, потому что его библиотека расширений столь велика. Возможно, это так же несправедливо, как заявление о том, что файл

/bin/sh раздут, поскольку велика коллекция всех сценариев в системе. Emacs можно было бы считать виртуальной машиной или конструкцией вокруг коллекции небольших, точных инструментов (режимов), которые были написаны на Lisp.

С такой точки зрения главное отличие между shell и Emacs заключается в том, что дистрибьюторы Unix не поставляют вместе с shell все существующие в мире сценарии. Неприязнь к Emacs из-за того, что наличие в нем универсального языка выглядит как распухание кода, приблизительно столь же неразумна, как отказ от использования сценариев ввиду того, что в shell имеются условные операции и циклы. Так же как для использования shell-сценариев не требуется изучать shell, нет необходимости изучать язык Lisp, для того чтобы пользоваться редактором Emacs. Если в Emacs имеется проблема проектирования, то она связана не столько с Lisp-интерпретатором (структурная часть), сколько с тем, что библиотека режимов представляет собой беспорядочное "нагромождение исторических наростов". Однако пользователи могут пренебречь данным источником сложности, так как неиспользуемые компоненты не влияют на их работу.

Этот аргумент весьма удобен. Его можно применять к другим конструкциям, интегрирующим инструментальные средства, таким как (неудобно большие) проекты настольных интегрированных пакетов GNOME и KDE. Что-то в нем притягивает. Кроме того, не следует доверять какой-либо "перспективе", предлагающей столь искусно разрешить все сомнения. Она может оказаться рационализацией и не решить основную проблему.

Поэтому необходимо избегать крайностей отрицания или принятия того, что Emacs является полезной и большой программой — это

и есть аргумент против минимализма Unix. Что кроме этого предлагает анализ видов сложности? И существует ли причина верить, что данные уроки сводятся к общим законам?

Для Unix-идеологии использования небольших, точных инструментов характерна скрытая двойственность; фон настолько неявный, что многие Unix-практики не замечают его или замечают не больше, чем рыба замечает воду, в которой плавает. Этим фоном являются интегрирующие структуры приложений (frameworks).

Небольшие точные инструменты в Unix-стиле сталкиваются с трудностями при совместном использовании данных, если они не находятся внутри структуры, облегчающей их взаимодействие. Emacs является примером такой структуры, а

единообразное управление общим контекстом достигается ценой необязательной сложности Emacs. Практическое влияние единообразного управления общим контекстом заключается в том, что пользователь не обременен проблемами низкоуровневого присваивания имен и управления ресурсами.

В Unix старой школы единственной интегрирующей структурой были конвейеры, перенаправление и shell. Интеграция осуществлялась с помощью сценариев, а общим контекстом была (по существу) сама файловая система. Однако на этом развитие не закончилось.

Программа Emacs объединила файловую систему с множеством текстовых буферов и вспомогательных процессов, почти совершенно оставляя структуру shell позади. Редактор Wily также предполагает использование буферов и вспомогательных процессов, но содержит в себе структуру shell. Современные настольные среды предоставляют GUI-интерфейсам структуру для обмена данными и тоже оставляют структуру shell позади. Каждая интегрирующая структура приложений имеет свои сильные и слабые стороны. Структуры приложений стали "жилищами" для множества инструментальных средств — shell для сценариев, Emacs для Lisp-режимов, а настольные среды для групп приложений, обменивающихся данными в GUI как посредством технологии "перетащить и отпустить", так и с помощь более "эзотерических" способов, таких как объектные брокеры.

Это выдвигает правило минимальности:

необходимо выбирать общий контекст, которым требуется управлять, и создавать настолько малые программы, насколько позволяют данные границы. То есть "так просто, как только возможно, но не проще", но основное внимание в данном случае уделяется выбору общего контекста. Правило применимо не только к структурам, но и к приложениям и программным системам.

Однако очень легко ошибиться при определении необходимых размеров общего контекста. Над разработчиком довлеет сила, выраженная законом Завински, - склонность приложений к совместному использованию контекста в целях удобства использования. Вовсе нетрудно, в конце концов, получить чрезмерный размер, слишком большое количество предположений и писать большие программы с излишней сложностью. Так, URL mailto: (типичный пример 1990-х годов) вызвал рост количества крупных почтовых клиентов, встроенных в Web-браузеры.

Средство исправления данной тенденции приходит прямо из "священных книг" Unix — правило экономии:

писать большую программу следует только в том случае, когда после демонстрации становится ясно, что ничего другого не остается, т.е. когда попытки расчленить проблему были предприняты, но потерпели неудачу. Данный принцип предполагает строгий скептицизм

в отношении больших программ и стратегию, позволяющую избежать их создания: прежде всего, следует искать решение на основе небольшой программы. Если одна небольшая программа не решает задачу, необходимо попытаться создать инструментарий, состоящий из небольших взаимодействующих программ внутри существующей структуры. И только в том случае, если оба подхода оказались безуспешными, традиции Unix позволяют разработчику создать крупную программу (или новую структуру приложения), не чувствуя себя при этом побежденным сложностью проектной задачи.

При написании интегрирующей структуры приложений необходимо помнить правило разделения. Структуры приложений должны быть механизмом и заключать в себе как можно меньше политики. В большинстве случаев вообще без политики. Необходимо выделять как можно больше поведения в модули, которые используют структуру приложений. Одним из преимуществ написания или повторного использования структуры является то, что она позволяет выделить код, который в противном случае состоял бы из огромных монолитов политики, в отдельные модули, режимы или инструментальные средства, т.е. блоки, которые можно полезно комбинировать с другими.

Данные правила эмпирические, однако конфликт в сердце Unix-традиции не может быть разрешен несколькими

априорными рецептами по оптимальному размеру любого заданного проекта. Обстоятельства изменяют решения, а гармония правильного мышления и хорошего стиля — основная задача разработчиков программного обеспечения. Аналогично тому, как в Дзэн путешествие

является целью, "просветления" приходится каждый раз достигать заново путем повседневной практики.

Часть III

Реализация

14

Языки программирования: C или не C?

Границы моего языка — границы моего мира.

Логико-философский трактат

(Tractatus Logico-Philosophicus 5.6, 1918) —Людвиг Виттгенштейн (Ludwig Wittgenstein).

14.1. Многообразие языков в Unix

В Unix поддерживается более широкий по сравнению с любой другой операционной системой диапазон языков прикладного программирования. Фактически Unix способна поддерживать больше различных языков, чем все вместе взятые операционные системы в истории вычислительной техники[117].

Существует по крайней мере две весомые причины столь значительного разнообразия. Во-первых, широкое использование операционной системы Unix как исследовательской и обучающей платформы. Второй причиной (гораздо более значимой для работающих программистов) является тот факт, что подбор подходящего языка (или языков) для конструкции приложения может привести к огромным различиям в продуктивности программиста. Следовательно, Unix-традиции поддерживают проектирование узкоспециальных языков (как было сказано в главах 7 и 9) и языков, которые в настоящее время обычно называются

языками сценариев , разработанных специально для связывания между собой других приложений и инструментальных средств.

Понятие "язык сценариев" (scripting language), вероятно, происходит от термина "сценарий" (script), который применялся к заранее подготовленному вводу для программы, обычно работающей в интерактивном режиме, в частности, sh или ed — гораздо более уместный термин, чем "runcom", унаследованный Unix от предка, операционной системы CTSS. Слово "script" появляется в руководстве для системы V7 (1979). Я не помню, кто именно придумал это название. Дуг Макилрой.

В действительности термин "язык сценариев" несколько неудобен. Многие из основных языков, обычно описываемых как языки сценариев (Perl, Tcl, Python и другие), уже переросли первоначальные задачи создания сценариев и в настоящее время являются самостоятельными универсальными языками программирования значительной мощности. Данный термин склонен срывать сильное сходство в стиле с другими языками, которые обычно не причисляются к этой группе, особенно с Lisp и Java. Единственным аргументом, оправдывающим нынешнее использование данного понятия, является тот факт, что лучшего термина еще никто не придумал.

Частично причиной того, что данные языки можно объединить в группу "языков сценариев" является то, что все они имеют почти совершенно идентичный онтогенез. Наличие динамической среды для интерпретации также сравнительно облегчает автоматизацию управления динамической памятью. Автоматизация управления динамической памятью требует использования ссылок (трудных для понимания адресов памяти, которые разработчик не в состоянии вычислять) вместо распространения копий значений или явных указателей. Использование ссылок делает динамический полиморфизм и ОО-методики следующим простым этапом.

Для того чтобы эффективно применять философию Unix, инструментарий программиста должен включать в себя не только C. Программисту потребуется изучить использование некоторых других языков в Unix (особенно языков сценариев), а также способы удобного сочетания нескольких языков, каждый из которых играет особую роль в крупных программных системах.

В данной главе рассматривается язык C и его наиболее важные альтернативы, обсуждаются их сильные и слабые стороны, а также виды задач, которым они наилучшим образом соответствуют. Ниже описываются языки C, C++, shell, Perl, Tcl, Python, Java и Emacs Lisp. Каждый обзорный раздел включает в себя учебные примеры приложений, написанных с использованием данных языков, а также ссылки на другие примеры и учебные материалы. Высококачественные реализации всех данных языков доступны в Internet в виде открытого исходного кода.

Внимание: выбор языка прикладного программирования является одним из основных идеологических вопросов в сообществе Internet/Unix. Люди сильно привязываются к данным средствам и иногда защищают их вопреки здравому смыслу. Если глава достигнет своей цели, то вполне может оскорбить фанатичных приверженцев всех языков, однако все остальные разработчики почерпнут из нее немало полезного.

14.2. Доводы против C

C — естественный язык операционной системы Unix. С начала 1980-х годов он стал доминировать в системном программировании почти повсеместно в компьютерной индустрии. За пределами сокращающейся ниши Fortran в научных и инженерных вычислениях, а также исключая невидимую массу финансовых приложений на языке COBOL в банках и страховых компаниях, C и его потомок C++ уже более десяти лет доминируют в прикладном программировании.

Поэтому утверждение о том, что C и C++ почти всегда являются неподходящим связующим материалом для начала разработки новых приложений, может показаться неверным. Тем не менее, оно справедливо. C и C++ оптимизируют машинную эффективность ценой увеличения времени реализации и (особенно) отладки. Несмотря на то, что писать на C или C++ системные программы и чувствительные ко времени выполнения ядра приложений все еще имеет смысл, мир значительно изменился со времен возвышения этих языков в 1980-х годах. Сегодня процессоры в тысячи раз быстрее, модули памяти в тысячи раз больше, а диски в

десять тысяч раз больше, причем примерно по тем же ценам[118].

Падение цен фундаментально изменило экономику программирования. В большинстве обстоятельств уже не имеет смысла так экономить аппаратные ресурсы, как это позволяет C. Напротив, экономически оптимальный выбор состоит в минимизации времени отладки и максимизации возможности долгосрочного сопровождения кода. Следовательно, большинство видов реализации (включая создание прототипов приложений) лучше обслуживаются более новым поколением интерпретируемых языков и языков сценариев. Эта трансформация точно соответствует тем условиям, которые на предыдущем витке исторической спирали привели к восхождению C/C++ и снизили важность программирования на ассемблере.

Центральной проблемой C и C++ является то, что они требуют от программистов самостоятельно осуществлять управление памятью — объявлять переменные, явно управлять связными списками, определять размеры буферов, обнаруживать или предотвращать переполнение буферов, а также распределять и высвобождать динамическую память. Некоторые из указанных задач могут быть автоматизированы путем искусственных действий, таких как дополнение C программой сборки мусора, например, реализация Boehm-Weiser, однако конструкция C такова, что подобное решение не может быть совершенным.

Управление памятью в C — серьезный источник трудностей и ошибок. По оценкам одного исследования (цитата из [9]), 30 или 40% времени разработки отводится на управление памятью в программах, которые манипулируют сложными структурами данных. В это число даже не включаются затраты на отладку. Несмотря на отсутствие точных данных, многие опытные программисты уверены, что ошибки управления памятью являются единственным крупнейшим источником постоянных ошибок в реальном коде[119]. Переполнение буфера является обычной причиной аварий и брешей в системе безопасности. Управление динамической памятью особенно чревато порождением коварных и трудно отслеживаемых

ошибок, таких как утечки памяти и проблемы недействительного указателя.

Вместе с тем не так давно ручное управление памятью имело смысл. Однако теперь "малых систем" больше нет, а в передовом программировании приложений ручное управление памятью не требуется. В современных условиях гораздо более целесообразно использовать язык реализации, который автоматизирует управление памятью (и на порядок сокращает количество ошибок ценой использования несколько большего числа циклов и памяти).

В недавней статье [63] собран впечатляющий массив статистических данных в пользу заявления, которое опытные программисты сочтут весьма правдоподобным: продуктивность программистов при работе с языками сценариев почти в два раза больше продуктивности при работе с C или C++. Данное утверждение хорошо согласуется с приведенной выше оценкой затрат времени (30-40%), а ведь еще следует учесть издержки отладки. Потери производительности при использовании какого-либо языка сценариев очень часто незначительны для реальных программ, поскольку такие программы склонны ограничиваться ожиданием I/O-событий, сетевой задержки и заполнением кэша, а не эффективностью, с которой они используют сам процессор.

В действительности, сообщество Unix медленно приближается к данной точке зрения, особенно с 1990 года, это видно по возрастающей популярности Perl и других языков сценариев. Однако развитие практики еще (к середине 2003 года) не привело к крупномасштабным переменам. Многие Unix-программисты до сих пор осмысливают урок, преподаваемый языками Perl и Python.

Та же тенденция, хотя и выраженная не так ярко, наблюдается за пределами мира Unix, например, в продолжающемся переходе от C++ к Visual BASIC, который заметно проявляется в разработке приложений для Microsoft Windows и NT, а также в движении к Java в мире мэйнфреймов.

Аргументы против C и C++ в равной степени применимы к другим традиционным компилируемым языкам, таким как Pascal, Algol, PL/I, FORTRAN и компилируемые диалекты BASIC. Несмотря на отдельные "героические усилия", такие как Ada, отличия между традиционными языками остаются внешними при сопоставлении их основных конструктивных решений, оставляющих управление памятью программисту. В Unix большинство из когда-либо созданных языков доступны в виде высококачественных реализаций с открытым исходным кодом. Несмотря на это, в широком использовании в Unix или Windows не осталось других традиционных языков. Разработчики отказались от них в пользу C или C++. Соответственно, в данной главе они не рассматриваются.

14.3. Интерпретируемые языки и смешанные стратегии

Языки с автоматическим управлением памятью осуществляют его с помощью диспетчера памяти, встроенного в их динамически исполняемые модули. Как правило, среды выполнения в таких языках разделены на программную часть (собственно выполняющийся сценарий) и часть интерпретатора с управляемой им динамической памятью. В Unix-системах (а также в других современных операционных системах) память интерпретатора может совместно использоваться несколькими программными частями, что сокращает фактические издержки для каждой из них.

Использование сценариев — нисколько не новая идея в мире Unix. В 1970-х годах, в эпоху гораздо меньших машин, Unix shell (интерпретатор для команд, вводимых в Unix-консоль) был спроектирован как полностью интерпретируемый язык программирования. Даже тогда

было распространено написание программ полностью на shell или использование shell для написания связующей логики, объединяющей встроенные утилиты и нестандартные программы на C в целостные системы, эффективность которых была больше, чем сумма эффективности составляющих частей. В классических вводных книгах по Unix-среде (таких как

"The Unix Programming Environment" [39]) подробно рассматривается данная тактика, и это вполне обосновано: она была одной из важнейших нововведений операционной системы Unix.

В современном shell-программировании языки свободно смешиваются, для решения подзадач задействуются как бинарные, так и интерпретируемые элементы из почти десятка других языков. Каждый язык решает ту задачу, к которой он приспособлен лучше остальных, каждый компонент является модулем с узкими интерфейсами для связи с другими модулями, а глобальная сложность системы в целом гораздо ниже, чем в случае ее реализации в виде одного массивного монолита на универсальном языке программирования.

14.4. Сравнение языков программирования

Сочетание языков представляет собой стиль программирования, который требует скорее более интенсивного использования знаний, чем программного кода. Для успешной работы разработчику необходимо владеть достаточным количеством языков программирования, а также знать преимущества каждого из них и иметь опыт их совместного использования. В этом разделе содержатся ссылки, которые помогут в этом. Для каждого рассмотренного языка приводятся учебные примеры, иллюстрирующие его достоинства.

14.4.1. C

Несмотря на проблему управления памятью, существуют прикладные области, в которых C остается наилучшим языком. C является оптимальным языком для программ, в которых нужна максимальная скорость, актуальны требования реального времени или необходима тесная связь с ядром операционной системы.

Язык C также является оптимальным для программ, переносимых на многие операционной системы. Однако рассмотренные ниже альтернативы C все больше используются в основных операционных системах, отличных от Unix; в ближайшем будущем значение переносимости как основного преимущества C может уменьшаться.

Иногда преимущество, которое достигается с помощью таких существующих программ, как генераторы синтаксических анализаторов или GUI-построители, которые генерируют код C, так велико, что оправдывает программирование на C остальной части небольшого приложения.

И конечно, C доказал свою незаменимость для разработчиков всех его альтернатив. При достаточно глубоком изучении реализации каждого из рассматриваемых здесь языков обнаруживается ядро, выполненное на простом, переносимом C. Эти языки унаследовали многие из преимуществ C.

В современных условиях, возможно, лучше было бы рассматривать C как ассемблер

высокого уровня для виртуальной машины Unix (учебный пример в главе 4). В другие операционные системы C-стандарты внесли такие возможности этой виртуальной машины, как стандартная библиотека ввода-вывода. Язык C незаменим, когда требуется тесная связь с аппаратной составляющей при сохранении переносимости.

Еще одна весомая причина, по которой следует изучать C (даже если требования к языку программирования разработчика удовлетворяются языком более высокого уровня), состоит в том, что это способствует развитию мышления на уровне аппаратной архитектуры. Лучшим справочником и учебным пособием по C для программистов является книга

"The C Programming Language" [42].

Перенос C-кода из одного вида операционных систем Unix в другой почти всегда возможен и обычно прост, но в некоторых областях, таких как сигналы и контроль над процессами, могут возникать определенные сложности. Некоторые из этих проблем рассматриваются в главе 17. Различие C-привязок в других операционных системах, несомненно, может вызвать проблемы переносимости C, хотя операционная система Windows NT, по крайней мере, теоретически должна поддерживать ANSI/POSIX-совместимый стандарт C API.

Высококачественные компиляторы C доступны в Internet в виде программ с открытым кодом; наиболее известным и широко применяемым является компилятор C Фонда Свободного программного обеспечения (Free Software Foundation — FSF), входящий в коллекцию компиляторов GNU (GNU Compiler Collection — GCC). Данный компилятор стал базовым для всех Unix-систем с открытым кодом, а также для некоторых систем с закрытым кодом. Версии GCC доступны даже для операционных систем семейства Microsoft. Исходные коды GCC доступны на сайте FSF <ftp://ftp.gnu.org/pub/gnu>.

Подводя итог, можно отметить, что преимуществом C является эффективность использования ресурсов и близость к аппаратной составляющей, а к недостаткам следует отнести то, что программисту, использующему C, приходится самостоятельно решать проблемы управления ресурсами.

14.4.1.1. Учебный пример:

fetchmail

Наилучший пример использования C — само ядро Unix, для которого язык программирования, свободно поддерживающий операции на аппаратном уровне, действительно является сильным преимуществом. Но

fetchmail представляет собой пример пользовательской утилиты, которую хорошо писать на C.

fetchmail совершает простейшие действия по управлению динамической памятью. Единственная сложная структура данных программы состоит из однонаправленного списка управляющих блоков, по одному на каждый почтовый сервер, который составляется только один раз при запуске и довольно незначительно изменяется впоследствии. Это значительно ослабляет довод против использования C, обходя основной недостаток языка.

С другой стороны, эти управляющие блоки достаточно сложны (они включают в себя строки, флаги и численные данные) и их трудно было бы обрабатывать как связанные объекты быстрого доступа на языке реализации без эквивалентной C функции структуры.

Большинство альтернатив C уступают ему в этом отношении (за исключением Python и Java).

Наконец,

fetchmail нуждается в возможности анализировать довольно сложный синтаксис спецификации для управляющей информации по каждому почтовому серверу. В Unix такая проблема классически решается с помощью генераторов C-кода, вырабатывающих исходный код для лексического и грамматического анализатора из декларативных спецификаций. Существование

yacc и

lex было доводом в пользу C.

fetchmail можно было бы на приемлемом уровне реализовать на Python, хотя и со значительной потерей производительности. Размер и сложность структуры данных

fetchmail полностью исключили бы shell и Tcl и строго указывали бы на недопустимость использования Perl. При этом предметная область программы находится за пределами естественных возможностей Emacs Lisp. Реализация на языке Java не была бы лишена смысла, но объектно-ориентированный стиль и уборка мусора дали бы небольшой выигрыш в специфических проблемах

fetchmail по сравнению с теми, которые уже решались с помощью C. C++ так же не смог бы значительно упростить сравнительно простую внутреннюю логику

fetchmail .

Однако реальная причина написания

fetchmail на C состоит в том, что программа создавалась как постепенное развитие ее предшественника, уже написанного на C. Существующая реализация на C прошла обширное тестирование на разнообразных платформах, на необычных и специфических серверах. Повторная реализация на другом языке была бы сложной и запутанной. Более того,

fetchmail зависит от импортируемого кода для функций (таких как NTLM-аутентификация), которые очевидно не доступны на более высоком уровне, чем C.

Интерактивный конфигуратор

fetchmail, лишенный традиционной проблемы C, написан на Python; соответствующий учебный пример рассматривается далее.

14.4.2. C++

Когда в середине 1980-х годов C++ был впервые выпущен в свет, объектно-ориентированные языки программирования были широко разрекламированы как радикальное средство против сложности программного обеспечения. Объектно- ориентированные возможности C++ казались бесспорным преимуществом над C, и приверженцы ожидали, что C++ быстро сделает своего предшественника устаревшим.

Этого не случилось. Отчасти причинами могут быть проблемы в самом C++; требование обратной совместимости с C привело ко многим компромиссам в конструкции. Кроме всего

прочего, данное требование помешало перевести C++ на полностью автоматическое управление динамической памятью и решить самую серьезную проблему C. Позже "гонка вооружений" между различными разработчиками компиляторов, которую не сдерживала слабая и преждевременная попытка стандартизации, привела к тому, что C++ стал "витиеватым" и чрезвычайно сложным языком.

Причиной послужило также и то, что самой OO-технологии не удалось оправдать ожидания. Проблема рассматривалась в главе 4 — OO-методы склонны приводить к созданию больших связующих уровней и проблемам сопровождения. На сегодняшний день, изучение архивов открытого исходного кода (где выбор языка скорее отражает суждения разработчика, чем корпоративные указания) показывает, что использование C++ до сих пор в основном характерно для GUI, мультимедийного инструментария и игр (основных областей успеха OO-конструкций), и гораздо реже встречается в других областях.

Возможно, что реализация OO-технологии на C++ особенно склонна к созданию проблем. Существуют данные, которые говорят о том, что расходы на жизненный цикл программ на C++ больше, чем у эквивалентов, написанных на C, FORTRAN или Ada. Является ли данная проблема проблемой OO, проблемой C++, или это комплексная проблема — пока не ясно, хотя есть смысл подозревать последнее [34].

В последние годы C++ включил в себя некоторые важные не-OO-идеи. В языке существуют исключения, подобные исключениям в Lisp, т.е. можно добавлять объект или значение в стек вызовов до тех пор, пока его не перехватит обработчик. STL (Standard Template Library, стандартная библиотека шаблонов) обеспечивает типовое программирование, т.е. возможность писать код алгоритмов, независимых от сигнатур их данных, и компилировать их для выполнения необходимых функций во время выполнения. (Это необходимо только для языков, выполняющих статический контроль типов в процессе компиляции; более динамические языки просто передают ссылки на переменные без типа и поддерживают идентификацию типов во время выполнения.)

Эффективный компилируемый язык, совместимость снизу вверх с языком C, объектно-ориентированная платформа, связующий материал для самых современных методик, таких как STL и типовое программирование, — C++ претендует на роль универсального средства, но ценой этого является сложность, гораздо большая, чем та, с которой может справиться мозг отдельного программиста. Как было отмечено в главе 4, главный конструктор языка признал, что он и не ждет, что какой-либо один программист сможет постичь язык полностью. Unix-хакеры не слишком хорошо отреагировали на сложность C++. Подтверждением этого служит одно из анонимных, но известных определений: "C++ — это спрут, полученный путем пришивания лишних ног собаке".

Однако фундаментальная проблема заключается в том, что по существу C++ является просто еще одним традиционным языком. Он лучше сдерживает проблему управления памятью, чем до создания стандартной библиотеки шаблонов (STL), и значительно лучше, чем C. Однако это довольно сомнительное решение проблемы, и оно не будет надежно работать, если в коде не используются объекты и только объекты. Возможности OO для многих типов приложений незначительны и просто добавляют сложности в C, не достигая весомых преимуществ. Доступны компиляторы C++ с открытым исходным кодом; если бы C++ определенно превосходил C, то в настоящее время он бы доминировал.

Подводя итоги, можно сделать вывод, что преимуществом C++ является сочетание эффективности компилируемого языка со средствами объектно-ориентированного и типового программирования. К недостаткам относятся его вычурность и сложность, а также склонность к поддержке сверхсложных конструкций.

Использование C++ целесообразно в том случае, когда существующий инструментарий или

служебные библиотеки C++ предлагают значительные преимущества для приложения или когда предметная область приложения лежит в упомянутых выше областях, где использование ОО-языка, несомненно, является большим преимуществом.

Классическим справочником по C++ является книга Страуструпа (Stroustrup)

"The C++ Programming Language" [82]. Отличными учебным пособием по C++ и основным ОО-методам для начинающих является книга

"C++: A Dialog" [36]. Статья

"C++ Annotations" [6] представляет собой сжатый вводный курс по C++ для опытных программистов на C.

Компилятор C++ входит в состав коллекции компиляторов GNU. Поэтому, язык доступен как для Unix, так и для операционных систем Microsoft; в данном случае также применимы комментарии, касающиеся C. Доступна обширная коллекция вспомогательных библиотек с открытым исходным кодом <http://www.boost.org/>. Однако переносимость языка ограничена тем, что (в середине 2003 года) в действующих реализациях C++ реализованы значительно отличающиеся подмножества проектного ISO-стандарта, который в настоящий момент находится в процессе подготовки[120].

14.4.2.1. C++ учебный пример: инструментарий Qt

Интерфейсный инструментарий Qt представляет собой замечательный пример успеха C++ в современном мире программ с открытым исходным кодом. Инструментарий предоставляет комплект элементов управления, а также API для написания графических пользовательских интерфейсов для системы X Window, который был сознательно (и довольно эффективно) разработан таким образом, чтобы имитировать вид и восприятие интерфейсов Motif, MacOS Platinum или Microsoft Windows. Фактически Qt предоставляет гораздо больше, чем просто GUI-службы, — он предоставляет переносимый прикладной уровень с классами для XML, доступа к файлам, сокетов, параллельных процессов, таймеров, обработки даты и времени, доступа к базам данных, различных абстрактных типов данных и Unicode.

Инструментарий Qt является важным и видимым компонентом проекта KDE, старейшего из двух конкурирующих проектов с открытым исходным кодом по разработке GUI и интегрированного набора настольных приложений.

Реализация Qt на C++ демонстрирует преимущества ОО-языка программирования для инкапсуляции компонентов пользовательского интерфейса. В языке, поддерживающем объекты, видимая иерархия элементов управления интерфейса может быть четко выражена с помощью иерархии экземпляров классов в коде программы. Несмотря на то, что то же самое можно сымитировать в C с помощью явного преобразования через созданную вручную таблицу методов, код, написанный на C++, будет гораздо чище. Полезно сравнить его с известным своей причудливостью C API в Motif.

Исходный код и справочная документация по Qt доступна на сайте Trolltech <http://www.trolltech.com/>.

14.4.3. Shell

Первым (и долгое время единственным) переносимым интерпретируемым языком был 'Bourne shell' (sh) 7 версии Unix. В настоящий момент родоначальный Bourne shell в значительной степени вытеснили разновидности совместимого снизу вверх Korn Shell (ksh). Единственным наиболее важным из них является Bourne Again Shell (bash).

Существуют и согласованно используются несколько других видов shell, но как языки программирования они не столь значительны. Наиболее известным из них, вероятно, является C shell (csh), печально известный своей непригодностью для написания сценариев [121].

Элементарные shell-программы чрезвычайно просты и естественны в написании. Язык shell послужил началом Unix-традиции быстрого создания прототипов на интерпретируемых языках.

Я написал самую первую версию netnews в виде shell-сценария в 150 строк. Она имела несколько групп новостей и перекрестную рассылку. Группы новостей представляли собой каталоги, а перекрестная рассылка была реализована как множественные ссылки на статью. Программа была слишком медленной для реального использования, но ее гибкость позволяла бесконечно экспериментировать с конструкцией протокола. Стивен М. Белловин.

Однако по мере увеличения своего размера, shell-программы часто становятся скорее узкоспециальными. Некоторые моменты синтаксиса shell (особенно правила использования кавычек и синтаксис операторов) могут сбить с толку. Данные недостатки, как правило, обусловлены компромиссами в той части конструкции shell, которая связана с языком программирования. Эти компромиссы были внесены, для того чтобы сохранить эффективность shell как интерактивного интерпретатора командной строки.

Программы описывают как "shell-программы" даже если они не написаны исключительно на shell, но в них интенсивно используются C-фильтры, такие как

sort(1) , и стандартные мини-языки обработки текста, такие как

sed(1) или

awk(1) . Однако данный вид программирования в течение нескольких лет идет на убыль. В наши дни подобная сложная связующая логика обычно пишется на Perl или Python, а shell резервируется для простейших упаковщиков (для которых данные языки были бы чрезмерно сложными) и сценариев инициализации системы (которая не предполагает, что они доступны).

Базовое shell-программирование достаточно описано в любой вводной книге по Unix.

"The Unix Programming Environment" [39] остается одним из лучших источников для программистов среднего и высокого класса. Реализации или клоны Korn shell представлены в каждой Unix-системе.

Сложные shell-сценарии часто имеют проблемы переносимости не столько из-за самой оболочки, а ввиду сделанных в них предположений о доступности той или иной программы в качестве компонента. Несмотря на то, что в отдельных случаях клоны bash и ksh доступны в операционных системах, отличных от Unix, shell-программы (практически) вообще невозможно перенести за пределы Unix.

В завершение отметим, что лучшее качество shell — естественность и быстрота написания небольших сценариев. Худшей стороной shell является то, что крупные shell- сценарии

зависят от множества вспомогательных команд, которые вовсе не обязательно идентично ведут себя и даже не всегда присутствуют на всех целевых машинах. Кроме того, анализировать зависимости в крупных shell-сценариях также непросто.

Почти никогда не возникает необходимость компилировать или устанавливать shell, поскольку все Unix-системы и эмуляторы Unix поставляются с подобными оболочками. Стандартной оболочкой для Linux и других передовых вариантов Unix в настоящее время является bash.

14.4.3.1. Учебный пример: xmlto

xmlto — управляющий сценарий, который вызывает все команды, необходимые для представления какого-либо XML-DocBook-документа в виде HTML, PostScript, простого текста или любого другого доступного формата (более подробно DocBook рассматривается в главе 18). Сценарий xmlto написан в bash.

Детали вызова XSLT-процессора обрабатываются в xmlto с помощью соответствующей таблицы стилей, затем результаты передаются постпроцессору. Для HTML и XHTML вся обработка сводится к трансформации. Для получения простого текста XML также трансформируется в HTML, а затем передается постпроцессору —

lynx(1) в режиме -dump, который преобразовывает HTML в простой текст. Для получения PostScript XML трансформируется в XML FO (Formatting Objects — объекты форматирования), которые постпроцессор затем преобразовывает в TEX-макрос, DVI-формат посредством

tex(1) , а затем, наконец, в PostScript с помощью широко известного инструмента

dvi2ps(1) .

xmlto состоит из одного интерфейсного shell-сценария. Он вызывает одну из нескольких подключаемых подпрограмм, каждая из которых названа по имени целевого формата. Каждая подключаемая подпрограмма представляет собой shell-сценарий. В зависимости от того как она вызвана, подключаемая подпрограмма либо предоставляет клиентской части таблицу стилей, либо вызывает соответствующий постпроцессор (или постпроцессоры) с различными заранее заданными аргументами.

Такая структура означает, что вся информация о заданном целевом формате находится в одном месте (соответствующем подключаемом сценарии), поэтому добавление новых целевых типов можно осуществить, не нарушая интерфейсного кода вообще.

Программа xmlto является хорошим примером shell-приложения среднего размера. Использование C или C++ не имело бы смысла, поскольку они неудобны для написания сценариев. Для такой программы можно было бы использовать любой из языков сценариев, описанных в данной главе; однако программа состоит только из простых команд управления без внутренних структур данных или сложной логики, поэтому достаточно использовать shell, так как это дает значительное преимущество — повсеместное распространение на целевых системах.

Теоретически данный сценарий можно было бы выполнить на любой системе, поддерживающей bash. Единственным ограничением является необходимость присутствия в системе одного из XSLT-процессоров, а также всех постпроцессоров. На практике маловероятна работа данного сценария во всех системах, кроме современных Unix-систем с

открытым исходным кодом.

14.4.3.2. Учебный пример: Sorcery Linux

Sorcerer GNU/Linux — дистрибутив Linux, устанавливаемый как небольшая, загрузочная опорная система, мощность которой достаточна для работы

bash(1) и нескольких утилит загрузки данных. Имея этот код, можно вызвать Sorcery, систему пакетов Sorcerer.

Sorcery обеспечивает установку, удаление и проверку целостности пакетов программ. Когда пользователь "ввел заклинания", Sorcery загружает исходный код, компилирует, инсталлирует его, а затем сохраняет список установленных файлов (наряду с протоколом компиляции и контрольными суммами для всех файлов). Установленные пакеты можно "отклонить" или удалить. Списки пакетов и проверка целостности также доступны. Более подробно данный дистрибутив описан на сайте проекта <http://sorcerer.wox.org>.

Система Sorcery полностью написана в shell. Процедуры инсталляции — небольшие, простые программы, для которых shell является оптимальным выбором. В данном конкретном случае главный недостаток shell нейтрализуется, так как авторы дистрибутива могут гарантировать, что необходимые вспомогательные программы будут присутствовать в опорной системе.

14.4.4. Perl

Perl — shell на стероидах. Данный язык был специально предназначен для замены

awk(1) и расширен, чтобы заменить shell в качестве уровня, связывающего сценарии, написанные на нескольких языках. Первая версия Perl вышла в 1987 году.

Самым сильным качеством Perl являются его чрезвычайно мощные встроенные средства для шаблонной обработки текстовых, строковых форматов данных, и в этой области Perl является непревзойденным. Данный язык также включает в себя гораздо более прочные структуры данных, чем shell, включая динамические массивы элементов различных типов, а также тип "hash" или "dictionary", который поддерживает удобный и быстрый поиск пар имя-значение.

Дополнительно Perl включает в себя довольно полную и хорошо продуманную привязку практически всего API-интерфейса Unix, что радикально уменьшает потребность в C и делает Perl пригодным для таких задач, как простые TCP/IP-клиенты и даже серверы. Другим серьезным преимуществом Perl является то, что вокруг него сформировалось крупное и жизнеспособное сообщество открытого исходного кода. Центром данного сообщества в сети является ресурс Comprehensive Perl Archive Network (обширны архив Perl-программ) <http://www.cpan.org>. Преданные Perl-хакеры создали сотни свободно используемых Perl-модулей для множества различных задач программирования. В число данных модулей входит структурный обход дерева каталогов, X-инструментарий для создания GUI-интерфейсов, превосходные встроенные средства для поддержки HTTP-роботов и CGI-программирования.

Главным недостатком Perl является то, что его части неисправимо уродливы, сложны и для

их использования требуется особая осторожность и стереотипные способы (соглашения о передаче аргументов функциям являются хорошим примером всех трех проблем). Начать работать с Perl сложнее, чем с shell. Хотя мелкие программы на Perl могут быть чрезвычайно мощными, необходима четкая дисциплина для поддержки модульности и сохранения конструкции под контролем по мере роста размера программы. Поскольку некоторые ограничивающие конструкторские решения в ранней истории Perl невозможно было переделать, многие из более сложных функций имеют довольно хрупкую конструкцию.

Наиболее полным справочником по Perl является книга

"Programming Perl" [88]. В ней содержится почти вся необходимая информация по данному языку, однако книга печально известна своей плохой организацией; чтобы найти необходимые сведения, читателю приходится перерабатывать множество материала. Более упорядоченным источником является книга

"Learning Perl" [76].

Язык Perl универсален в Unix-системах. Perl-сценарии с одинаковым основным номером версии часто легко переносимы между Unix-системами (при условии, что в них не используются модули расширения). Реализации Perl доступны (и даже хорошо документированы) для операционных систем семейства Microsoft, а также MacOS. Perl/Tk обеспечивает кроссплатформенную GUI-совместимость.

Выводы: наибольшим преимуществом Perl является мощный аппарат для мелких связующих сценариев, включающих большой объем обработки регулярных выражений; наибольшим недостатком данного языка является то, что он уродлив, "несговорчив" и в больших объемах почти не поддерживается.

14.4.4.1. Небольшой учебный пример по Perl: blq

blq-сценарий представляет собой средство для опроса блок-списков (списки Internet-узлов, которые идентифицируются как постоянные источники нежелательных почтовых сообщений, известных также как спам). Текущую версию исходного кода можно найти на странице проекта blg <http://www.unicom.com/sw/blg/>.

blq — хороший пример небольшого Perl-сценария, иллюстрирующий как сильные, так и слабые стороны языка. В нем интенсивно используется средство обработки регулярных выражений. С другой стороны, используемый в сценарии модуль расширения Perl Net::DNS необходимо включать в зависимости от обстоятельств, поскольку не гарантируется, что он присутствует в какой-либо заданной Perl-инсталляции.

Сценарий blq, как Perl-код, является исключительно четким и организованным, рекомендуется изучить его как пример хорошего стиля (хорошими примерами также являются другие Perl-инструменты, упомянутые на сайте проекта blq). Однако некоторые части кода покажутся нечитабельными, если не знать весьма специфических идиом Perl — в качестве примера можно привести первую строку кода $0=~s!.*/!!;. Хотя все языки характеризуются некоторой непрозрачностью, непрозрачность Perl — худшая из всех.

Для написания небольших сценариев данного типа хорошо подходят языки Tcl и Python, однако оба они испытывают недостаток удобных функций Perl для поиска регулярных выражений, которые интенсивно используются в blq. Реализации рассматриваемой программы на Tcl или Python были бы приемлемыми, но, вероятно, менее компактными и

выразительными. Написание подобной программы на Emacs Lisp отняло бы даже меньше времени, и программа была бы более компактной, но, вероятно, чрезвычайно медленной в работе.

14.4.4.2. Большой учебный пример по Perl: keeper

Программа keeper — инструмент для создания картотеки поступающих пакетов и поддержки FTP- и index-файлов WWW для крупных архивов программного обеспечения Linux на сайте проекта ibiblio. Исходный код и документацию можно найти в подкаталоге поисковых инструментов архива ibiblio <http://www.ibiblio.org>.

Программа keeper — хороший пример интерактивного Perl-приложения более чем среднего размера. Интерфейс командной строки является строчным и создан по образцу специализированной оболочки или редактора каталогов; необходимо отметить встроенные справочные возможности. Рабочие части интенсивно используют обработку файлов и каталогов, поиск шаблонов и шаблонное редактирование. Следует отметить легкость, с которой keeper генерирует Web-страницы и email-уведомления из программных шаблонов. Кроме того, примечательно использование заготовленного Perl-модуля для автоматизации обхода различных функций в дереве каталогов.

Данное приложение, состоящее примерно из 3300 строк кода, вероятно, превышает рекомендуемый при создании одной Perl-программы предел размера и сложности. Тем не менее, большая часть keeper была написана в течение шести дней. Для написания такой программы на C, C++ или Java потребовалось бы шесть недель, а программа была бы крайне трудной в отладке или модификации. Для чистого Tcl данный путь слишком сложен. Версия на Python, вероятно, была бы структурно более чистой, читаемой и удобной в обслуживании, но также и более хаотичной (особенно в части поиска шаблонов). Задачи данной программы можно было бы выполнять с помощью какого-либо режима Emacs Lisp, но Emacs не подходит для использования на telnet-канале, который часто затормаживается из-за перегрузок сервера.

14.4.5. Tcl

Tcl (Tool Command Language — язык инструментальных команд) — небольшой языковой интерпретатор, предназначенный для связи с компилируемыми C-библиотеками и обеспечивающий управление C-кодом с помощью сценариев (

extended scripts — расширенные сценарии). Первоначально он применялся для управления библиотеками для электронных симуляторов (SPICE-подобных приложений). Tcl также подходит для

встроенных сценариев (embedded scripts) , т.е. сценариев, которые вызываются из C-программ и возвращают им значения. Первая общедоступная версия Tcl вышла в 1990 году.

Некоторые средства, созданные на основе Tcl, широко используются за пределами Tcl-сообщества. Ниже описываются два наиболее важных из них.

• Инструментарий Tk, более податливый и легкий X-интерфейс, который упрощает быстрое

создание кнопок, диалоговых окон, меню и полос прокрутки текста, а также собирает входные данные от них.

• Expect, язык, который сравнительно упрощает включение в сценарии полностью интерактивных программ с большим разнообразием ответов.

Инструментарий Tk настолько важен, что язык часто называется Tcl/Tk. Tk также часто используется с языками Perl или Python.

Главным преимуществом языка Tcl является то, что он чрезвычайно гибок и в своей основе прост. Его синтаксис необычен (основывается на позиционном анализаторе), но полностью последователен. В языке отсутствуют зарезервированные слова, а также синтаксическое различие между вызовом функции и "встроенным" синтаксисом. Таким образом, можно из Tcl эффективно переопределить собственно языковой интерпретатор (что делает рациональными проекты, подобные Expect).

Основным недостатком Tcl является то, что чистый язык имеет только слабые средства для управления пространством имен и модульностью, а два из них (upvar и uplevel) довольно опасны, если использовать их неосторожно. Также не существует структур данных, кроме ассоциативных списков. Поэтому Tcl расширяется очень слабо — трудно организовывать и отлаживать чистую Tcl-программу даже умеренного размера (более нескольких сотен строк), так как можно запутаться в собственном коде. На практике почти во всех Tcl-программах используется одно из нескольких OO-расширений для языка.

Необычность синтаксиса на первых порах также может оказаться проблемой. В течение какого-то времени разработчику сложно разобраться в различиях между строковыми кавычками и скобками, а правила заключения строк в скобки или кавычки несколько сложные.

Чистый Tcl предоставляет доступ только к сравнительно небольшой и широко используемой части Unix API (по существу, только обработка файлов, создание подпроцессов и сокетов). Действительно, Tcl интересен как демонстрация того, каким небольшим и вместе с тем полезным может быть язык сценариев. Tcl-расширения (подобно модулям Perl) обеспечивают более развитый набор возможностей, однако (как и в случае CPAN-модулей) нет гарантии, что они установлены везде.

Оригинальным справочником по Tcl является книга

"Tcl and the Tk Toolkit" [58], но ее уже почти "затмила" книга

"Practical Programming in Tcl and Tk" [89]. Брайан Керниган создал описание реального Tcl-проекта [38], в котором показаны преимущества и недостатки Tcl как инструмента быстрого создания прототипов и реальных программ. Его противопоставление Microsoft Visual BASIC особенно взвешенно и поучительно.

Сообщество Tcl не имеет одного центрального репозитария, поддерживаемого основной группой, аналогичной группам поклонников языка Perl или Python, но несколько выдающихся Web-сайтов ссылаются друг на друга и описывают большую часть разработки Tcl-инструментов и расширений. Прежде всего, стоит обратить внимание на проект Tcl Developer Xchange <http://www.tcltk.com>; среди прочего он предоставляет Tcl-код интерактивного учебного пособия по Tcl. Кроме того, существует лаборатория Tcl на сайте SourceForge <http://sourceforge.net/foundry/tcl-foundry/>.

Tcl-сценарии имеют проблемы переносимости, аналогичные проблемам сценариев shell. Сам язык хорошо переносим на другие платформы, но компоненты, которые он вызывает, могут не быть таковыми. Существуют реализации Tcl для операционных систем семейства Microsoft, MacOS и многих других платформ. Tcl/Tk-сценарии будут работать на любой

платформе, обладающей возможностями поддержки GUI-интерфейса.

Выводы: сильной стороной Tcl является его экономная, компактная конструкция и расширяемость Tcl-интерпретатора; худшей стороной языка является необычный позиционный синтаксический анализатор, слабость структур данных и управления пространством имен. Последний дефект делает язык слабо масштабируемым для больших проектов.

14.4.5.1. Учебный пример:

TkMan

TkMan — браузер для man-страниц Unix и документов Texinfo. Исходный код состоит примерно из 1200 строк, что довольно много для программы, написанной на чистом Tcl, но код необычайно хорошо организован и разделен на модули. Для создания GUI-интерфейса, несколько лучшего, чем интерфейс, поддерживаемый стандартными утилитами

man(1) или

xman(1) , используется инструментарий Tk.

TkMan представляет собой хороший учебный пример, поскольку он демонстрирует почти полную гамму технических приемов Tcl. В число наиболее ярких входит Tk-интеграция, управление другими Unix-приложениями (такими как поисковая машина Glimpse) с помощью сценариев, а также использование Tcl для синтаксического анализа Texinfo-разметки.

Любой другой язык не предоставил бы настолько прямой интерфейс к Tk GUI, который составляет большую часть данного кода.

Исходный код и документацию можно найти в Web с помощью ключевого слова "TkMan".

14.4.5.2. Moodss: большой учебный пример по Tcl

Система Moodss представляет собой графическое мониторинговое приложения для системных администраторов. Оно способно следить за журналами и накапливать статистические данные по работе MySQL, Linux, SNMP-сетей и Apache, а также предоставляет их систематизированный обзор посредством GUI-панелей, которые имеют вид электронных таблиц и называются инструментальными панелями (dashboards). Модули мониторинга могут быть написаны на Python или Perl, а также на Tcl. Код безукоризненно написан, организован и считается примером для подражания в Tcl-сообществе. Существует Web-сайт проекта — <http://jfontain.free.fr/moodss/>.

Ядро программы состоит из 18 000 строк Tcl-кода. В нем используется несколько Tcl-расширений, включая специальную объектную систему. Автор Moodss признает, что без них "написание такого большого приложения было бы невозможным".

Как и в предыдущем случае, ни один из других языков не предоставил бы такого непосредственного интерфейса к Tk GUI, который составляет большую часть кода.

14.4.6. Python

Python является языком написания сценариев, предназначенным для тесной интеграции с C. Он способен как импортировать, так и экспортировать данные в динамически загружаемые C-библиотеки, его также можно вызывать из C как встроенный язык сценариев. Синтаксис Python очень напоминает гибрид синтаксиса C и семейства Modula, но имеет ту необычную особенность, что структура блока фактически определяется отступами (в языке нет аналогов явных директив begin/end или фигурных скобок C). Первая публичная версия Python вышла в 1991 году.

Язык Python представляет собой очень четкую, элегантную конструкцию с превосходными свойствами модульности. Он предоставляет разработчикам возможность писать программы в объектно-ориентированном стиле, но не вынуждает делать этого (возможно программирование более традиционным, процедурным способом, подобным C). Язык имеет систему типов, сравнимую по выразительности с системой типов Perl, включая динамические контейнеры и ассоциативные списки, но менее характерную (документально подтвержден тот факт, что объектная система Perl была создана как подражание системе Python). Данный язык устраивает даже Lisp-хакеров, предоставляя безымянные лямбда-операторы (задаваемые функциями объекты, которые могут распространяться и использоваться итераторами). Python поставляется с Tk-инструментарием, который можно использовать для простого создания GUI-интерфейсов.

Стандартный дистрибутив Python включает в себя клиентские классы для большинства важных Internet-протоколов (SMTP, FTP, POP3, IMAP, HTTP) и порождающие классы для HTML. Поэтому рассматриваемый язык хорошо подходит для протокольных роботов и каналов сетевого администрирования. Он также превосходно подходит для работы в Web CGI и успешно конкурирует с Perl на участке высокой сложности в данной прикладной области.

Из всех рассматриваемых интерпретируемых языков Python и Java являются двумя наиболее точно подходящими для масштабирования в большие сложные проекты с множеством взаимодействующих разработчиков. Во многом Python проще, чем Java, и его предрасположенность к быстрому созданию прототипов может дать ему преимущество перед Java для автономного использования в приложениях, которые не являются ни чрезмерно сложными, ни требующими высокой скорости выполнения. Реализация Python на языке Java, предназначенная для облегчения смешанного использования обоих языков, доступна и используется в реальных проектах. Она называется Jython.

Python не может конкурировать с C или C++ по скорости выполнения (хотя использование стратегии смешанных языков на сегодняшних быстрых процессорах, возможно, делает этот недостаток сравнительно второстепенным). Фактически Python, как правило, считается наименее эффективным и самым медленным из основных языков сценариев — это цена, которую приходится платить за полиморфизм типов во время выполнения. Однако не стоит по этой причине отказываться от использования данного языка. Большинству приложений фактически не требуется производительность, большая, чем предоставляется Python. Приложения, в которых такая производительность действительно необходима, обычно ограничены внешними задержками, такими как задержки сети или дисков. Кроме того, в виде компенсации, Python исключительно просто комбинируется с C, поэтому для достижения существенного прироста скорости чувствительные к производительности Python-модули можно легко преобразовывать в C.

Python проигрывает в выразительности языку Perl для небольших проектов и связующих сценариев, сильно зависящих от возможностей обработки регулярных выражений. Данный язык был бы избыточным в мелких проектах, для которых более подходящим может быть shell или Tcl.

Как и Perl, Python имеет хорошо организованное сообщество разработчиков с центральным Web-сайтом <http://www.python.org>, содержащим множество полезных реализаций Python, инструментов и модулей расширения.

Наиболее полным справочником по языку Python является книга

"Programming Python" [51]. Обширная документация по Python-расширениям также доступна на Web-сайте проекта Python.

Python-программы в большинстве случаев хорошо переносятся между Unix-системами и даже на другие операционные системы. Стандартная библиотека достаточно мощная для того, чтобы значительно сократить использование непереносимых вспомогательных программ. Реализации Python доступны для операционных систем Microsoft и MacOS. Кроссплатформенная разработка GUI возможна с помощью либо Tk, либо двух других инструментариев. Python/C-приложения могут быть "замороженными", квази-скомпилированными в чистый исходный код C, который должен быть переносимым на системы без установленного Python.

Выводы: лучшей стороной Python является то, что он поддерживает четкий, читаемый код и комбинирует доступность с расширяемостью в крупных проектах; наибольший недостаток данного языка заключается в его неэффективности и медлительности не только по сравнению с компилируемыми языками, но и относительно других языков сценариев.

14.4.6.1. Небольшой учебный пример по Python: imgsizer

Imgsizer — утилита, которая переписывает WWW-страницы так, что теги включения изображений автоматически получают верные размеры изображения (это ускоряет загрузку страниц во многих браузерах). Исходные коды и документацию можно найти в подкаталоге "URL WWW tools" <http://www.ibiblio.org> архива ibiblio.

Программа imgsizer первоначально была написана на Perl, и была почти идеальным примером небольшого, управляемого шаблонами инструмента обработки текста. Позднее программа была транслирована в Python в целях получения преимущества библиотеки Python по поддержке HTTP-передачи. В результате этого зависимость от внешней утилиты доставки страницы была устранена. Примечательно использование утилит

file(1) и

identify(1) пакета ImageMagick в качестве специальных средств для получения размеров изображений в пикселях.

Динамическая обработка строк и необходимость замысловатой проверки регулярных выражений сделала бы написание imgsizer на C или C++ крайне сложным. Такая версия также была бы значительно больше и сложнее для чтения. Язык Java решил бы проблему неявного управления памятью, но он едва ли был бы более выразительным, чем C или C++ при проверке текстовых шаблонов.

14.4.6.2. Учебный пример по Python среднего размера:

fetchmailconf

В главе 11 пара

fetchmail/fetchmailconf рассматривалась как пример одного из способов отделения реализации от интерфейса. Преимущества Python хорошо иллюстрируются на примере утилиты

fetchmailconf .

В программе

fetchmailconf используется инструментарий Tk для реализации многопанельного редактора конфигурации с графическим интерфейсом (для GTK также существуют Python-привязки и другие инструментарии, но Tk-привязки поставляются с каждым интерпретатором Python).

В режиме для высококвалифицированных пользователей GUI-интерфейс поддерживает редактирование около 60 атрибутов, распределенных по трем уровням панелей. Элементы управления атрибутами включают в себя комбинацию из флажков, переключателей, текстовых полей и списков. Несмотря на эти сложности, автору книги потребовалось меньше недели, включая четыре дня на изучение Python и Tk, для проектирования и кодирования первой полнофункциональной версии.

Python превосходит другие языки в быстром создании прототипов GUI-интерфейсов, а (как показывает пример

fetchmailconf) прототипы часто являются основными составляющими реальных проектов. Perl и Tcl имеют аналогичные сильные стороны в данной области (включая инструментарий Tk, который был написан для Tcl), но они сложнее в управлении на уровне сложности кода

fetchmailconf (приблизительно 1400 строк). Emacs Lisp не подходит для GUI-программирования.

Выбор Java увеличил бы издержки сложности программной задачи, не создавая значительных преимуществ для данного приложения, которое не требует высокой скорости выполнения.

14.4.6.3. Большой учебный пример Python: PIL

Python-библиотека обработки графики (Python Imaging Library — PIL) поддерживает обработку растровой графики. Программа поддерживает множество популярных форматов, включая PNG, JPEG, BMP, TIFF, PPM, XBM и GIF. Python-программы могут использовать ее для того, чтобы конвертировать и трансформировать изображения. В число поддерживаемых трансформаций входит кадрирование, вращение, масштабирование и сдвиг. Также поддерживается пиксельное редактирование, искривление изображений и преобразования цветов. Дистрибутив PIL включает в себя Python-программы, которые делают доступными данные средства из командной строки. Таким образом, программу PIL можно использовать для трансформации изображений в пакетном режиме или в качестве мощного инструментария, посредством которого реализуется программируемая обработка растровых

изображений.

Реализация PIL иллюстрирует способ, который позволяет легко расширять Python с помощью загружаемых расширений Python-интерпретатора, состоящих из объектного кода. Код библиотеки, реализующей основные операции над растровыми объектами, написан на C для обеспечения скорости. Более высокие уровни и программная логика, написанные на Python, работают медленнее, но проще для чтения, модификации и расширения.

Было бы сложно или даже невозможно написать аналогичный инструментарий на Emacs Lisp или shell, которые вообще не имеют или не документируют интерфейс C-расширений. Tcl имеет хорошие средства C-расширений, но программа PIL была бы неудобно большим проектом на Tcl. Подобные средства имеются в Perl (Perl XSO), но они представляют собой узкоспециальные решения, слабо документированы, сложны и нестабильны по сравнению со средствами Python, а также используются редко. Интерфейс машинно-зависимых методов в Java (Native Method Interface), вероятно, обеспечивает возможности, сравнимые с возможностями Python. Было бы, вероятно, рационально создать проект PIL на Java.

Код программы PIL и документация доступны на Web-сайте проекта <http://www.pythonware.com/products/pil/>.

14.4.7. Java

Язык программирования Java был разработан под девизом "написанное однажды работает везде" и предназначен для поддержки встроенных интерактивных программ (или

аплетов) на Web-страницах, которые запускались бы из любого браузера. Благодаря серии грубых технических и стратегических просчетов правообладателя, корпорации Sun Microsystems, данный язык потерпел неудачу в достижении обеих первоначальных целей. Однако Java до сих пор достаточно силен как в системном, так и в прикладном программировании, чтобы составить конкуренцию C и C++. Язык Java был анонсирован в 1995 году.

Язык Java талантливо спроектирован для использования больших преимуществ автоматического управления памятью и меньших, но не менее важных, преимуществ

поддержки OO-конструкции, будучи при этом меньше по размеру и сложности, чем C++. В данном языке заметно сохранен C-подобный синтаксис, который удобен большинству программистов. Java включает в себя поддержку внешних вызовов к динамически загружаемому C и вызовов Java как встроенного языка из C. Не случайно, что Sun выполнила большую работу по созданию хорошей документации по Java, доступной в Web.

Недостатком Java можно считать то, что (по сравнению, например, с Python) некоторые части языка представляются чрезмерно сложными, а другие неразвиты. Видимость классов Java и правила явного определения блоков являются вычурными. Такие функции, как внутренние и неименованные классы, могут привести к очень запутанному коду. Отсутствие надежных методов деструкторов означает, что трудно гарантировать соответствующее управление другими ресурсами, кроме памяти, например, мьютексами и блокировкой файлов. Значительная часть средств операционной системы Unix, включая сигналы, опрос и выбор, не доступна из Java в стандартной поставке. Несмотря на то, что в Java имеются очень мощные I/O-средства, простое чтение текстовых файлов затруднено.

Наблюдается и сложная проблема с библиотеками, подобная "аду DLL" в Windows. В Java

отсутствует метод для управления различными версиями библиотек, что может привести к возникновению крупных проблем в таких средах, как серверы приложений, где сервер может поставляться с одной версией (например) XML-библиотеки, а приложение с другой (обычно более новой) версией. Единственной возможностью решения таких проблем является переменная среды CLASSPATH, источник хронических проблем внедрения.

Более того, Sun неразумно управляет Java как с политической, так и с технической точки зрения. Первый GUI-инструментарий Java, AWT был неорганизованной смесью, которую необходимо было в корне заменить. Отказ от стандартизации ECMA/ISO еще больше раздражал разработчиков, уже расстроенных положениями SCSL (Sun Community Source License — лицензия на исходный код в сообществе Sun). Ограничения данного документа препятствуют реализациям Java 1.2 с открытым исходным кодом и их спецификациям J2EE (Java 2 Enterprise Edition). Это дискредитирует исходные цели универсальной переносимости Java.

К сожалению, аплеты браузеров мертвы. Фактически их уничтожило решение Microsoft не поддерживать Java 1.2 в Internet Explorer. Однако язык Java, вероятно, нашел гарантированную нишу в экологии вычислений — "сервлеты", работающие внутри серверов Web-приложений. Java также все чаще используется для многих внутри корпоративных программных разработок, не связанных непосредственно с базами данных или Web-серверами. Данный язык становится главным конкурентом как для платформы Microsoft ASP/COM, так и для Perl CGI. Наконец, Java широко распространен и все больше используется как язык для обучающего, вводного программирования (роль, для которой он чрезвычайно хорошо подходит).

В целом объективно можно сказать, что Java превосходит C++ (который гораздо сложнее и имеет меньше средств для решения проблемы управления памятью) для всех задач, кроме системного программирования и большинства приложений, чувствительных к скорости выполнения. Опыт показывает, что Java-программисты менее склонны попадать в ловушку излишней OO-иерархии, чем программисты C++, хотя это остается значительной проблемой.

Как Java достигнет равновесия с другими рассмотренными здесь языками до сих пор не ясно и может сильно зависеть от масштаба проекта. Можно ожидать, что соответствующая Java ниша будет подобна нише Python. Как и Python, Java не может конкурировать ни с C или C++ в отношении чистой скорости выполнения, ни с Perl в мелких проектах, где интенсивно используется шаблонное редактирование. Данный язык (более явно, чем Python) является избыточным для мелких проектов. Можно предположить, что Python получит преимущество в мелких проектах, а Java в более крупных, но окончательный результат будет ясен позднее.

Лучшим печатным справочником, вероятно, является книга

"Java In A Nutshell" [19], она, однако, не является лучшим учебным пособием. Таким, вероятно, можно считать книгу

"Thinking in Java" [17]. Ссылки на все мировые Web-сайты, посвященные Java, начинаются с сайта Java Sun <http://java.sun.com>, на котором также представлена полная HTML-документация, доступная для бесплатной загрузки. Полезные ссылки по Java также собраны на сайте проекта Open Directory Java Page <http://dmoz.org/Computers/Programming/Languages/Java/>.

Реализации Java доступны для всех Unix-систем, операционных систем Microsoft, MacOS и многих других платформ.

Исходные коды Kaffe, открытой реализации Java, в которой библиотеки классов согласуются с большей частью JDK 1.1 и частично с JDK 1.2, доступны на сайте проекта Kaffe <http://www.kaffe.org/>.

Существует Java-интерфейс для GCC. Программа GCJ способна компилировать Java-код либо в Java-байткод, либо в собственный код, а также Java-байткод — в собственный код. Данный компилятор поставляется в пакете с открытыми библиотеками классов, реализующими большую часть JDK 1.2, и интерпретатором Java-байткода, который называется

gij . Подробности описаны на странице проекта GCJ <http://gcc.gnu.org/java/>.

Существует Java IDE-среда для Emacs, см. сайт проекта JDEE <http://jdee.sunsite.dk/>.

Java прекрасно переносится между платформами на уровне языка. Проблема может возникнуть из-за несовершенных реализаций библиотек (особенно ранних версий JDK 1.1, которые не поддерживают более новый пакет JDK 1.2).

Наилучшим качеством Java является то, что данный язык достаточно плотно приблизился к идеалу "написанное однажды работает везде", для того чтобы быть полезным в качестве независимой от операционной системы среды. Наибольший недостаток Java заключается в том, что дробление Java 1/Java 2 подрывает эту цель, глубоко разочаровывая разработчиков.

14.4.7.1. Учебный пример: FreeNet

Freenet — одноранговый сетевой проект, цель которого заключается в том, чтобы сделать невозможной цензуру и подавление информационного наполнения Web-страниц[122]. Разработчики Freenet предвидят несколько видов использования проекта.

• Неподконтрольное цензуре распространение спорной информации: Freenet защищает свободу слова, без цензуры разрешая анонимную публикацию различных материалов, от рядовой альтернативной журналистики до официально запрещенных разоблачений.

• Эффективное распространение информационного наполнения, требующего большой полосы пропускания: адаптивное кэширование и зеркала используются для распространения программных обновлений Debian Linux.

• Всеобщая публикация персональных материалов: Freenet позволяет иметь Web-сайт любому желающему без ограничения пространства или принудительной рекламы, даже если предполагаемый Web-мастер не имеет компьютера.

Freenet достигает этих целей путем обеспечения виртуального пространства для публикации документов, которое не связано с какой-либо определенной машиной. Опубликованная информация и внутренние индексы внутренних данных Freenet растиражированы и распространены в сети так, что даже администраторы проекта не знают, где в определенный момент времени расположены все физические копии. Конфиденциальность просмотра или подписки на информацию Freenet защищена с помощью строгой криптографии.

Язык Java был хорошим выбором для данного проекта, как минимум, по двум причинам. Во-первых: цели проекта серьезно оценивают совместимые реализации на самых разнообразных машинах, поэтому высокая переносимость Java является главным преимуществом. Во-вторых: природа проекта такова, что важным является сетевой API-интерфейс, а Java имеет мощный встроенный API.

Язык C является традиционным для инфраструктурных проектов такого рода, которые имеют высокие требования к производительности, но недостаток стандартизированного сетевого

API значительно затруднил бы переносимость. Такие же трудности возникли бы и в случае использования C++. Языки Tcl, Perl или Python могли бы сократить нагрузку переносимости, но ценой больших потерь в производительности. Emacs Lisp был бы крайне медленным и совершенно несоответствующим.

14.4.8. Emacs Lisp

Emacs Lisp является языком сценариев, который используется для программирования режимов работы текстового редактора Emacs. Первая общедоступная версия данного языка появилась в 1984 году.

Emacs Lisp не является универсальным языком, как другие рассмотренные в данной главе языки. Несмотря на то, что он теоретически достаточно мощный, чтобы использовать его в качестве универсального, традиционно он применялся только для написания управляющих программ для самого редактора Emacs. Кроме того, Emacs Lisp не сообщается с другим программным обеспечением так же свободно, как современные языки сценариев.

Тем не менее, существует значительный диапазон применений, в которых Emacs более эффективен, чем любой другой язык. Многие из вариантов применения с обеспечением клиентского интерфейса для инструментов разработки, таких как C-компилятор и линкер, утилита

make(1) , системы контроля версий и символические отладчики; такое применение Emacs Lisp рассматривается в главе 15.

В более общем смысле, Emacs для шаблонного или синтаксического

интерактивного редактирования является тем же, чем язык Perl для шаблонного

пакетного редактирования. Любое приложение, в котором задействована интерактивная обработка специального файлового формата или текстовой базы данных, является достойным кандидатом для того, чтобы его прототип (а возможно, и окончательная реализация) был написан как Emacs-режим (программа на Emacs Lisp, которая точно определяет поведение редактора).

Emacs Lisp также является ценным для создания приложений, которые необходимо тесно интегрировать с каким-либо текстовым редактором или которые функционируют, главным образом, как программы просмотра с некоторыми возможностями редактирования. В эту категорию попадают пользовательские агенты для почтовых программ и Usenet-новостей, а также определенные виды внешних интерфейсов к базам данных.

Emacs Lisp — по сути Lisp. Следовательно, он автоматически управляет памятью и гораздо более изящен, а также имеет большую мощность, чем большинство традиционных языков, или, в действительности, большинство

нетрадиционных языков. Emacs Lisp на этом уровне способен соперничать с Java или Python и превосходит языки C или C++, Perl, shell или Tcl. Неизменная проблема Lisp, которая заключается в недостатке стандартизированной ОС-привязки для переносимости, решается ядром Emacs, которое на самом деле

является ОС-привязкой.

Другая непреодолимая проблема — чрезмерное потребление ресурсов, на современных

машинах более не является реальной проблемой. Ранее были широко распространены насмешливые расшифровки названия, такие как "Emacs Makes A Computer Slow" (Emacs тормозит компьютер) и "Eventually Munches All Computer Storage" (в конечном итоге занимает всю память) (фактически в дистрибутив Emacs включен список таких расшифровок). Но в настоящее время многие другие широко используемые категории программ (такие как Web-браузеры) превышают размеры и являются более сложными, чем Emacs, который в сравнении с ними кажется довольно умеренным.

Наиболее полным справочником по Emacs Lisp является руководство

"The GNU Emacs Lisp Reference Manual", которое можно просмотреть с помощью справочной системы "info-документов", встроенной в Emacs. Данное руководство также можно загрузить с FTP-узла FSF <ftp://ftp.gnu.org/pub/gnu>. Если оно покажется малопонятным, может помочь книга

"Writing GNU Emacs Extensions" [30].

Программы Emacs Lisp отличаются превосходной переносимостью между платформами. Реализации Emacs доступны для всех Unix-систем, операционных систем Microsoft и MacOS.

Выводы: наилучшим качеством Emacs Lisp является то, что он комбинирует отличный базовый язык Lisp с мощными примитивами для манипуляций с текстом; худшим качеством данного языка является его слабая производительность и трудности при обмене данными с другими программами.

Дополнительная информация, касающаяся

Emacs , приводится в разделе "Полезные сведения о Emacs" следующей главы.

14.5. Тенденции будущего

В таблице 14.1 приведены приблизительные показатели современного распространения языков. Данные взяты с проектов SourceForge[123] и Freshmeat[124], двух важнейших сайтов новых программ по состоянию на март 2003 года.

Таблица 14.1. Выбор языков Язык SourceForge Freshmeat 01.03.03 01.03.04 01.03.03 01.03.04 C 10296 13479 4845 6191 C++ 9880 13570 2098 2922 Shell 1058 1473 487 620 Perl 4394 5560 2508 3031 Tcl 649 811 328 377 Python 2222 3267 948 1459 Java 8032 12126 1900 2977 Emacs Lisp ? ? 31 37

Данные проекта SourceForge несколько сглажены по ряду причин: интерфейс запроса не позволяет осуществлять фильтрацию одновременно по операционным системам и языкам, поэтому некоторые из чисел содержат MacOS- и Windows-проекты. В результате, вероятно, значительно преувеличивается распространение C++ и Java. Однако Unix-проекты значительно преобладают (в соотношении приблизительно 3:1), поэтому показатели других языков, вероятно, не слишком искажены.

Данные Freshmeat меньше, но на данном сайте поддерживаются только Unix-версии программ и учитываются только актуальные разработки, а не беспорядочное нагромождение исчезнувших и приостановленных проектов SourceForge. Таким образом, примечательно, что совокупные данные отстают от данных SourceForge примерно в соотношении 1:2, кроме

случаев (C++ и Java), где можно было бы ожидать отклонения от пропорции ввиду отсутствия Windows-проектов.

Заметим, что черновик данной главы впервые был написан в 1997 году; в свет книга вышла в 2003 году. То есть прошло достаточно много времени, чтобы относительные позиции рассмотренных выше языков каким-либо образом изменились с момента первого описания и отразились тенденции выбора разработчиков. Эти тенденции указывают на то, что будущее языков будет очень похожим.

В целом, показатели C, C++ и Emacs Lisp оставались стабильными в течение 1997-2003 годов. Показатели языка C росли медленно за счет более ранних традиционных языков, таких как FORTRAN. С другой стороны, C++ утратил некоторую часть поклонников, которые предпочли язык Java.

Прилично возросло использование Perl, однако развитие языка приостановлено. Внутреннее устройство Perl печально известно своей неряшливостью. Несомненно, уже давно пора переписать реализацию данного языка с самого начала. Правда, одна такая попытка, предпринятая в 1999 году, провалилась, а другая пока, видимо, приостановлена. Тем не менее, Perl остается ведущим языком сценариев и преобладает в Web-сценариях и CGI.

Язык Tcl переживает период относительного спада. В 1996 году широко распространенная и правдоподобная оценка размеров сообщества указывала на то, что на каждого Python-хакера приходится пять Tcl-хакеров и двенадцать Perl-хакеров. По данным SourceForge, в настоящее время соотношение приблизительно равно 3:1:7. Однако Tcl представляется весьма широко используемым в написании сценариев для специализированных компонентов в различных отраслях промышленности, включая автоматизацию проектирования электроники, радио и телевещание, а также киноиндустрию.

Рост популярности Python настолько же стремителен, насколько стремителен спад Tcl. Хотя размеры Perl-сообщества до сих пор вдвое превышают численность поклонников Python, видимая тенденция перехода талантливейших Perl-хакеров к использованию Python является довольно угрожающей для первого языка, особенно, учитывая то, что миграция в противоположном направлении полностью отсутствует. Язык Java стал широко использоваться в тех местах, которые уже охвачены технологией Sun Microsystems, и активно внедряется в качестве учебного языка в образовательном процессе для студентов компьютерных специальностей. Вместе с тем в других областях данный язык только в малой степени более популярен, чем это наблюдалось в 1997 году. Стремление корпорации Sun к использованию частной модели лицензирования предотвратило главный рост, предсказываемый многими наблюдателями. В сообществе Linux и в более широком сообществе открытого исходного кода, язык Java не соперничает с C, как в других культурах.

За весь период не появилось ни одного нового универсального языка, который мог бы составить конкуренцию рассматриваемым здесь языкам. PHP вторгается в Web-разработку, вытесняя Perl CGI-сценарии (как и ASP и серверные Java-приложения), однако он почти никогда не используется для автономного программирования. Не-Emacs-диалекты Lisp, перспективная в свое время область, которая, казалось, должна была возродиться в середине 1990-х годов, продолжает сдавать свои позиции. Недавние разработки, такие как Ruby (гибрид Python-Perl-Smalltalk, разработанный в Японии) и Squeak (вариант Smalltalk с открытым исходным кодом), выглядят многообещающими, но пока не привлекли хакеров из других сообществ и не продемонстрировали "неослабевающей силы".

14.6. Выбор X-инструментария

Проблемой, связанной с выбором языка, является выбор X-инструментария для GUI-программирования. Здесь уместно упомянуть затронутую в главе 1 тему отделения политики от механизма в системе X.

Выбор X-инструментария связан с выбором прикладного языка по двум причинам: во-первых, поскольку некоторые языки поставляются с привязкой к предпочтительному инструментарию, а во-вторых, потому что некоторые виды инструментария имеют привязки только к ограниченному набору языков.

Несомненно, язык Java обладает собственными встроенными кроссплатформенными инструментариями, поэтому выбирать придется между AWT (используемым везде) и Swing (более мощным, более сложным, медленным и поставляемым только в составе пакета JDK 1.2/Java 2). В оставшейся части данного раздела основное внимание уделено другим уже рассмотренным языкам. Аналогично, при использовании Tcl будет использоваться и Tk. Вероятно, существует не слишком много особенностей в оценке альтернатив.

Некогда повсеместно используемый инструментарий Motif фактически вышел из употребления. Он был не способен держаться наравне с новыми инструментариями, распространяемыми без лицензионной платы или ограничений, которые привлекали внимание разработчиков до тех пор, п.ока не превзошли по возможностям и функциям своих предшественников с закрытыми исходными кодами. В настоящее время вся конкуренция сосредоточена в рамках движения открытого исходного кода.

В настоящее время серьезно стоит рассматривать четыре вида инструментария: Tk, GTK, Qt и wxWindows, из которых очевидно ведущими являются GTK и Qt. Для всех четырех инструментариев предусмотрены версии для MacOS и Windows, поэтому в любом случае разработчик получает возможность кроссплатформенной разработки.

Старейшим и наиболее распространенным из них считается инструментарий Tk. Он является собственным инструментарием для Tcl, и привязки к нему поставляются вместе со стандартной версией Python. Библиотеки для обеспечения языковых привязок к Tk, как правило, доступны для C и C++. К сожалению, стандартный набор элементов управления Tk ограничен и довольно уродлив. С другой стороны, элемент управления Canvas (холст) обладает возможностями, которые в других инструментариях до сих пор реализуются с трудом.

Инструментарий GTK возник как замена для Motif и создавался для поддержки GIMP. В настоящее время он является предпочтительным инструментарием проекта GNOME и используется в сотнях GNOME-приложений. Собственным API-интерфейсом является C. Доступны привязки для C++, Perl и Python, но они не поставляются в стандартных дистрибутивах языка. GTK является единственным из четырех инструментариев с естественной C-привязкой.

Qt — инструментарий, связанный с KDE-проектом. Он представляет собой собственную библиотеку C++. Доступны привязки для Python и Perl, но они не поставляются со стандартными интерпретаторами. Qt получил известность благодаря наличию хорошо спроектированного и наиболее выразительного API из всех четырех инструментариев, однако его принятие в начальной стадии было заблокировано полемикой по ранним версиям лицензии и в дальнейшем тормозилось медленным созданием C-привязки.

Инструментарий wxWindows также является естественным для C++ и имеет доступные привязки в Perl и Python. Его разработчики придают особое значение главным образом поддержке кроссплатформенной разработки и рассматривают ее как главную рыночную цель инструментария. Другая цель связана с тем, что wxWindows фактически является упаковщиком для собственных (GTK, Windows и MacOS 9) элементов управления на каждой

платформе, а поэтому приложения, написанные с его использованием, имеют естественные для данных систем вид и восприятие.

К середине 2003 года было описано не слишком много подробных исследований, однако Web-поиск фразы "X toolkit comparison" поможет найти некоторые полезные справочные сведения. В табл. 14.2 обобщена информация о состоянии рассмотренной области.

Таблица 14.2. Сравнительные характеристики X-инструментариев Инструментарий Собственный язык Поставляется с Привязки C C++ Perl Tcl Python Tk Tcl Tcl, Python + + + + + GTK C Gnome + + + + + Qt C++ KDE + + + + + wxWindows C++ – – + + + +

Архитектурно все данные библиотеки написаны на почти одном и том же уровне абстракции. В GTK и в Qt используются настолько подобные аппараты для обработки событий, что перенос программ между ними рассматривается как почти тривиальный. Выбор инструментария, вероятно, будет больше обусловлен доступностью привязок к используемому языку разработки, чем любыми другими факторами.

15

Инструментальные средства: тактические приемы разработчика

Unix дружественна к пользователю, но привередлива в выборе друзей. —Аноним

15.1. Операционная система, дружественная к разработчику

За операционной системой Unix давно закрепилась репутация хорошей среды для разработки программ. Она хорошо оснащена инструментами, написанными программистами для программистов. Данные инструменты автоматизируют многие рутинные мелкие задачи, которые в противном случае отвлекали бы внимание программиста от наиболее важного (и наиболее увлекательного) аспекта разработки — от проектирования.

Несмотря на то, что в Unix есть все необходимые инструменты и каждый из них хорошо документирован, они не связаны с помощью интегрированной среды разработки (Integrated Development Environment — IDE). Их поиск и внедрение в инструментальный набор, удовлетворяющий потребностям разработчика, всегда требовали значительных усилий.

Разработчику, привыкшему к хорошей IDE-среде (GUI-управляемой комбинации редактора, конфигуратора, компилятора и отладчика, которая в наши дни широко распространена в системах Macintosh и Windows), принятый в Unix подход может показаться бессистемным, туманным и примитивным. Однако в действительности он достаточно систематизирован.

Использование IDE имеет смысл для одноязыкового программирования в слабо оснащенной инструментами среде. Если работа программиста ограничена оттачиванием вручную кода на C или C++, то IDE-среды весьма целесообразны. Однако в Unix выбор языков и вариантов реализации гораздо разнообразнее, а практика использования нескольких генераторов кода,

специальных конфигураторов и многих других стандартных и нестандартных инструментов является общепринятой.

В Unix действительно существуют IDE-среды (имеется несколько таких сред с открытыми исходными кодами, включая эмуляции основных IDE Macintosh и Windows). Однако с их помощью трудно контролировать неограниченное множество инструментальных средств, и поэтому IDE-среды используются нечасто. Операционная система Unix поддерживает более гибкий стиль, центром которого не является исключительно цикл редактирование/компиляция/отладка.

В данной главе рассматриваются тактические приемы разработки в Unix — создание кода, управление его конфигурацией, профилирование, отладка, а также автоматизация большого количества монотонной работы, связанной с этими задачами, с тем чтобы разработчик мог сконцентрироваться на более увлекательных аспектах. Как обычно, при изложении материала основное внимание в большей степени уделено архитектурной картине, чем пошаговым инструкциям. Если же читатель

интересуется пошаговыми деталями, то рекомендуется обратиться к книге

"Programming with GNU Software" [50], в которой описывается большинство инструментов, рассмотренных в данной главе.

Многие из описываемых инструментов автоматизируют те работы, которые программист в состоянии выполнить самостоятельно и вручную, хотя и медленнее и с большим количеством ошибок. Однократные затраты на цикл обучения сполна окупятся способностью писать программы более эффективно и уделять меньше внимания низкоуровневым деталям и больше конструкции в целом.

Традиционно Unix-программисты учатся использовать данные инструменты у других программистов, а также в процессе многолетней практики. Начинающим программистам рекомендуется уделить особое внимание данной главе, поскольку в ней в сжатой форме приведен большой раздел обучающего цикла Unix путем демонстрации возможностей непосредственно на начальном этапе. Опытные программисты в случае нехватки времени могут пропустить данную главу, однако она может оказаться полезной и им, поскольку здесь могут встретиться такие полезные практические рекомендации, которые не известны даже им.

15.2. Выбор редактора

Первым и самым основным инструментом разработки является текстовый редактор, подходящий для модификации и написания программ.

В Unix доступны буквально десятки текстовых редакторов. Написание редактора, вероятно, является одним из стандартных практических упражнений для подающих надежды хакеров в сообществе открытого исходного кода. Большинство таких редакторов недолговечны, они не подходят для продолжительного использования кем-либо другим, кроме их авторов. Некоторые редакторы моделируют аналогичные не-Unix-программы, полезные в качестве вспомогательных переходных средств для программистов, привыкших к другим операционным системам. Широкий выбор текстовых редакторов доступен на сайте проекта SourceForge, ibiblio или в других основных архивах открытого исходного кода.

В качестве инструментов для серьезной работы в сфере Unix-программирования полностью

доминируют два редактора. Каждый из них доступен в нескольких вариантах реализации, однако имеет стандартную версию, которую, несомненно, можно найти в любой современной Unix-системе. Речь идет о редакторах

vi и

Emacs. Они рассматривались в главе 13 как часть темы целесообразного размера программного обеспечения.

Как отмечалось в главе 13, данные редакторы отражают тенденции резко контрастирующих философий проектирования, причем оба являются чрезвычайно популярными среди определенной части пользовательского контингента. Опросы Unix- программистов непротиворечиво указывают на соотношение 50/50 между ними, тогда как доля всех остальных редакторов минимальна.

Ранее при рассмотрении

vi и

Emacs основное внимание уделялось их необязательной сложности и сопутствующим вопросам философии проектирования. Многие другие аспекты данных редакторов достойны изучения как с практической точки зрения, так и с точки зрения культурной грамотности в Unix-сообществе.

15.2.1. Полезные сведения о

vi

Название

vi — аббревиатура от "visual editor" (визуальный редактор), произносится как

"ви ай" (а не

"вай" и

определенно не

"шесть").

vi не был самым ранним экранным редактором. Пальма первенства в этой области принадлежит программе Rand editor (re), которая работала в Version 6 Unix в 1970-х годах. Однако

vi — самый долгоживующий экранный редактор, созданный для Unix, который до сих пор используется и является "священной" составляющей традиции Unix.

Первоначальная версия

vi была в наличии в самых ранних дистрибутивах BSD начиная с 1976 года; в настоящее время она устарела. Данную версию заменил редактор "new vi", который поставлялся с 4.4BSD и имеется в современных ее вариантах, таких как BSD/OS, FreeBSD и NetBSD. Существует несколько вариантов с расширенными функциями, особенно

vim, vile, elvis и

xvi , среди которых

vim, вероятно, является наиболее популярным и поставляется в составе многих Linux-систем. Все варианты довольно похожи и используют основной набор команд, неизменный со времен первоначальной версии

vi.

Версии

vi доступны для операционных систем Windows и MacOS.

Большинство вводных книг по Unix включают в себя главу, описывающую основное использование редактора

vi. Ответы на часто задаваемые вопросы по использованию

vi доступны на сайте Editor FAQ/vi <http://www.faqs.org/faqs/editot-faq/vi/>. Множество других копий данной страницы можно найти с помощью поиска в Web страниц, в заголовках которых имеются слова "vi" и "FAQ".

15.2.2. Полезные сведения о Emacs

Emacs означает "EDiting MACroS" (произносится

"и-макс"). Он первоначально был написан в конце 1970-х годов как набор макросов в редакторе, который назывался TECO, после чего переписывался несколько раз различными способами. Забавно, что современные реализации Emacs включают в себя режим эмуляции TECO.

Ранее при обсуждении редакторов и необязательной сложности отмечалось, что многие пользователи считают Emacs чрезмерно тяжеловесным. Однако затраты времени на его изучение окупаются впоследствии повышением продуктивности. Emacs поддерживает множество мощных режимов редактирования, которые помогают с синтаксисом различных языков программирования и разметки. Далее в настоящей главе рассматривается возможность использования Emacs в комбинации с другими средствами разработки, предоставляющей возможности, сравнимые (а во многих случаях превосходящие) с возможностями традиционных IDE-сред.

Стандартной версией Emacs, повсеместно доступной на современных Unix-системах, является

GNU Emacs; программа, которая обычно запускается при вводе команды emacs в командной строке Unix-оболочки. Исходный код и документация по GNU Emacs доступны на сайте архива Фонда свободного программного обеспечения <ftp://gnu.org/pub/gnu>.

Существует вариант, который называется

XEmacs. Он имеет улучшенный X-интерфейс, но совершенно те же возможности (унаследованные от Emacs 19). Домашняя страница

XEmacs: <http://www.xemacs.org>. Emacs (и Emacs Lisp) повсеместно доступны в

современных Unix-системах. Он перенесен на MS-DOS (где работает слабо), а также на операционные системы Windows 95 и NT (где, как говорят, работает достаточно неплохо).

Emacs включает в себя собственное интерактивное учебное руководство и очень подробную документацию. Инструкции по запуску данных ресурсов можно найти на стандартном экране запуска Emacs. Хорошим введением является книга

"Learning GNU Emacs" [10].

Клавиатурные комбинации, используемые в Unix-версиях Netscape/Mozilla, а также в текстовых окнах Internet Explorer (в формах и почтовой программе), скопированы со стандартных привязок для основных операций редактирования текста. Данные привязки — ближайшие элементы к кроссплатформенному стандарту клавиатурных комбинаций редакторов.

15.2.3. "Антирелигиозный" выбор: использование обоих редакторов

Многие люди, обычно использующие оба редактора

vi и

Emacs, склонны применять их для различных задач и находят весьма ценными преимущества использования обоих.

Вообще,

vi наилучшим образом подходит для решения мелких задач — быстрого написания ответов на письма, простых изменений в конфигурации системы и т.д. Данный редактор особенно полезен при работе в новой системе (или на удаленной системе через сеть), когда не доступны файлы настроек Emacs.

Роль Emacs — расширенные сеансы редактирования, в которых необходимо решать сложные задачи, модифицировать несколько файлов и использовать результаты работы других программ в течение данного сеанса. Для программистов, использующих систему X на своих консолях (что типично для современных Unix-систем), считается обычным запускать Emacs сразу после регистрации в системе в большом окне и оставлять его работающим постоянно, возможно, просматривая десятки файлов и даже запуская программы в многочисленных подокнах Emacs.

15.3. Генераторы специализированного кода

Unix имеет давнюю традицию поддержки инструментов, которые специально предназначены для генерации кода для различных специальных целей. Давними "монументами" данной традиции, которые "уходят корнями" в Version 7 и ранние дни Unix, а также фактически использовались для написания оригинального Portable C Compiler в 1970-х годах, являются утилиты

lex(1) и

yacc(1) . Их современными совместимыми потомками являются

flex(1) и

bison(1) , часть GNU-инструментария, которая до сих пор интенсивно используется и в наши дни. Данные программы послужили примером, который продолжает развиваться в проектах, подобных построителю интерфейсов

Glade в GNOME.

15.3.1.

yacc и

lex

Программы

yacc и

lex являются инструментальными средствами для генерации синтаксических анализаторов языков программирования. В главе 8 отмечалось, что свой первый мини-язык программист часто создает случайно, а не как часть запланированной конструкции. В результате, как правило, появляется созданный вручную синтаксический анализатор, который приводит к чрезмерным затратам времени на сопровождение и отладку, особенно, если разработчик не поймет, что часть разработанного им кода является синтаксическим анализатором и не отделит его соответствующим образом от остальной части кода приложения. Генераторы синтаксических анализаторов являются инструментами, которые позволяют добиться большего, чем создание случайной, узкоспециальной реализации. Они не только позволяют разработчику выразить спецификацию грамматики на более высоком уровне, но и четко отделяют сложность реализации синтаксического анализатора от остального кода.

Если разработчик достиг того момента, когда планируется реализовать мини- язык с нуля, а не путем расширения или внедрения существующего языка сценариев или анализатора XML, то утилиты

yacc и

lex , вероятно, окажутся наиболее важными инструментами после компилятора C.

Как

lex , так и

yacc генерируют код для одной функции, соответственно, для "получения лексемы из входного потока" и для "синтаксического анализа последовательности лексем на предмет ее соответствия грамматике". Обычно созданная

yacc функция синтаксического анализатора вызывает функцию анализатора лексем, сгенерированного

lex , каждый раз при необходимости получения следующей лексемы. Если в yacc-сгенерированном синтаксическом анализаторе вообще не существует написанных пользователем обратных вызовов C, то работа данного анализатора сводится только к

проверке синтаксиса. Возвращаемое значение сообщит вызывающей программе о совпадении входных данных с ожидаемой грамматикой.

В более распространенном варианте пользовательский C-код, встроенный в сгенерированный синтаксический анализатор, заполняет некоторые динамические структуры данных как побочный эффект синтаксического анализа ввода. Если мини-язык является декларативным, то приложение может использовать эти структуры данных непосредственно. В случае императивного мини-языка структуры данных могут включать в себя дерево грамматического разбора, которое немедленно передается некоторой оценочной функции.

Утилита

yacc имеет довольно некрасивый интерфейс — через экспортируемые глобальные переменные с именным префиксом yy_. Это связано с тем, что программа

yacc предшествовала структурам C. Фактически

yacc предшествовала самому языку C; первая реализация утилиты была написана на языке B, предшественнике C. Грубый, хотя и эффективный алгоритм, используемый в сгенерированных

yacc синтаксических анализаторах при попытках восстановления после ошибок анализа (лексемы выталкиваются до тех пор, пока не обнаружится явная ошибка), также может привести к проблемам, включая утечки памяти.

Если вы создаете деревья грамматического разбора с использованием функции malloc и начинаете выталкивать элементы стека в процессе восстановления после ошибки, то вам не удастся восстановить (высвободить) память. Как правило, утилита

yacc не способна это делать, поскольку не имеет достаточных сведений о содержимом стека. Если бы yacc-анализатор был написан на C++, то он мог бы "предположить", что значения являются классами, и использовать деструктор. В "реальных" компиляторах узлы дерева грамматического разбора генерируются с помощью распределителя динамической памяти, поэтому узлы "не вытекают", но так или иначе, проявляется логическая утечка памяти, которую необходимо проанализировать, чтобы создать систему восстановления после ошибок промышленного уровня.Стив Джонсон.

Программа

lex — генератор лексических анализаторов. Она входит в состав того же функционального семейства, что и

grep(1) и

awk(1) , но является более мощной, поскольку позволяет подготовить произвольный C-код для выполнения при каждом совпадении. Программа принимает декларативный мини-язык и создает "скелетный" C-код.

Для того чтобы понять работу lex-сгенерированного анализатора лексем, существует грубый, но удобный способ — мысленная инверсия работы

grep(1) . Тогда как

grep(1) принимает одно регулярное выражение и возвращает список совпадений во входном потоке данных, каждый вызов lex-сгенерированного анализатора лексем принимает список регулярных выражений и указывает, какое выражение встречается следующим в потоке данных.

Разделение анализа ввода на распознавание лексем и синтаксический анализ потока лексем является полезным тактическим приемом, даже если в ходе разработки утилиты Yacc и Lex не используются, а "лексемы" не имеют ничего общего с обычными лексемами в компиляторе. Неоднократно я убеждался, что разделение обработки входных данных на два уровня значительно упрощает код и облегчает понимание, несмотря на сложность, внесенную самим разделением. Генри Спенсер.

Утилита

lex была написана для автоматизации создания лексических анализаторов (анализаторов лексем) для компиляторов. Впоследствии оказалось, что она имеет удивительно широкий диапазон применения для других видов распознавания образцов, и с тех пор описывается как "швейцарский нож Unix-программирования"[125].

При разрешении любой проблемы распознавания образцов или создания конечного автомата, в котором все возможные входные сигналы умещаются в байте, утилита

lex позволяет сгенерировать код, который будет более эффективным и надежным, чем созданный вручную конечный автомат.

Джон Джарвис (John Jarvis) в Холмделе (Holmdel — лаборатория AT&T) использовал

lex для поиска неисправностей в монтажных платах. Он сканировал плату, применял методику кодирования цепей для представления границ областей на плате, а затем использовал Lex для определения образцов, с помощью которых можно было бы находить распространенные ошибки монтажа.Майк Леск.

Важнее то, что мини-язык lex-спецификации является более высокоуровневым и компактным, чем эквивалентный C-код, написанный вручную. Доступны модули для использования

flex (версия с открытым исходным кодом) с Perl (их можно найти в Web с помощью фразы "lex perl"), а также идентично работающая реализация, которая является частью средства

PLY в Python.

lex генерирует синтаксические анализаторы, работающие на порядок медленнее написанных вручную. Однако данный факт не является причиной для ручного кодирования, это аргумент в пользу создания с помощью

lex прототипа и доработки кода вручную, только если прототип показывает реальное "бутылочное горлышко".

Утилита

yacc — генератор синтаксических анализаторов. Она также была написана для автоматизации части работы по написанию компиляторов,

yacc принимает на входе грамматическую спецификацию в декларативном мини-языке, подобном BNF (Backus-Naur Form — запись Бэкуса-Наура), с C-кодом, связанным с каждым элементом грамматики. Данная программа генерирует код для функции синтаксического анализа, которая при вызове принимает текст, соответствующий грамматике из входного потока. По мере распознавания каждого грамматического элемента, функция анализатора запускает связанный C-код.

Комбинация утилит

lex и

yacc весьма эффективна для написания языковых интерпретаторов всех видов. Хотя большинству Unix-программистов никогда не придется выполнять данный вид универсального построения компилятора, для которого задумывались эти инструменты, они чрезвычайно полезны для написания анализаторов синтаксиса конфигурационных файлов и узкоспециальных мини-языков.

Сгенерированные с помощью

lex анализаторы лексем работают очень быстро при распознавании низкоуровневых образцов во входных потоках, однако известный утилите

lex язык регулярных выражений плохо подходит для вычисления или распознавания рекурсивно вложенных структур. Для их анализа потребуется

yacc. С другой стороны, несмотря на то, что теоретически возможно написать yacc-грамматику с собственным сбором лексем, такая грамматика была бы перегружена кодом, а анализатор был бы крайне медленным. Для анализа входных лексем следует использовать

lex. Таким образом, данные инструменты являются симбиотическими.

Если существует возможность реализовать анализатор на языке более высокого уровня, чем C (что и рекомендуется; см. главу 14), то следует рассмотреть такие

эквивалентные средства, как PLY в Python (которое охватывает функции

lex и

yacc)[126] или Perl-модули PY и Parse::Yapp, либо Java-пакеты CUP[127], Jack[128] или Yacc/M[129].

Как и в случае с макропроцессорами, одной из проблем, связанных с генераторами кода и препроцессорами, является то, что ошибки компиляции в сгенерированном коде могут содержать номера строк сгенерированного кода (который редактировать нежелательно), а не номера строк во входных данных генератора (т.е. там, где необходимо внести изменения). В утилитах

yacc и

lex данная проблема решается такими же конструкциями #line, что и в препроцессоре C. Они устанавливают текущий номер строки для отчета об ошибках. Любая программа, генерирующая код на C или C++, должна работать аналогичным образом.

В более широком смысле хорошо спроектированные генераторы процедурного кода никогда не должны требовать от пользователя исправлять вручную или даже просматривать сгенерированный код. Создание корректного кода является непосредственной задачей генератора.

15.3.1.1. Учебный пример: грамматика fetchmailrc

Канонический демонстрационный пример, который, видимо, приводится в каждом учебном пособии по

lex и

yacc, представляет собой игрушечную программу интерактивного калькулятора, которая анализирует и вычисляет введенные пользователем арифметические выражения. В данной книге нет этого избитого клише. Заинтересованные читатели могут обратиться к исходному коду реализации

bc(1) и

dc(1) проекта GNU или к принципиальному примеру "hoc"[130] см. [39].

Вместо этого грамматика анализатора конфигурационных файлов

fetchmail предоставляет хороший учебный пример среднего размера по использованию

lex и

yacc . Здесь имеется несколько интересных моментов.

lex -спецификация в файле rcfile_l.l — весьма типичная реализация shell-подобного синтаксиса. Обратите внимание на то, как два дополняющих правила поддерживают строки либо с одинарными, либо с двойными кавычками; данная идея хороша в принципе. Правила для принятия (возможно, со знаком) целых литералов и отклонения комментариев также являются достаточно распространенными.

yacc- спецификация в файле rcfile_y.y достаточно длинная, но понятная. Она не осуществляет каких-либо

fetchmail -действий, а только устанавливает биты в списке внутренних управляющих блоков. После запуска

fetchmail в обычном режиме программа только периодически проходит по данному списку, используя каждую запись для управления сеансом получения почты с удаленного узла.

15.3.2. Учебный пример:

Glade

Программа

Glade рассматривалась в главе 8 в качестве хорошего примера декларативного мини-языка. Также отмечалось, что в результате работы серверной части

Glade генерируется код на одном из нескольких языков.

Glade представляет собой хороший современный пример генератора прикладного кода. Описанные ниже функции, которые отсутствуют в большинстве GUI-построителей (особенно в большинстве коммерческих GUI-построителей), делают

Glade. Unix-программой "по духу".

•

Glade GUI и генератор кода

Glade не связаны в массивном монолите, а подчиняются правилу разделения (и построены согласно модели "разделения ядра и интерфейса").

• GUI и генератор кода соединяются с помощью текстового формата (основанного на XML), который можно читать и модифицировать с помощью других инструментов.

• Поддерживается несколько целевых языков (а не только C или C++). Существует возможность легко добавлять другие языки.

Конструкция позволяет при необходимости заменить редактор GUI-интерфейса в

Glade .

15.4. Утилита

make : автоматизация процедур

Сами по себе исходные коды программ не делают приложения. Также важен способ их компоновки и упаковки для распространения. Операционная система Unix предоставляет инструментальное средство для частичной автоматизации данных процессов —

make(1) . Утилита

make описывается в большинстве вводных книг по операционной системе Unix. Более конкретная ссылка приводится в книге

"Managing Projects tenth Make" [57]. В случае использования

GNU make (наиболее развитого варианта

make, который обычно поставляется в составе Unix-систем с открытым исходным кодом) рецепты книги

"Programming with GNU Software" [50] могут в некотором отношении оказаться лучшими. Большинство Unix-систем, содержащих

GNU make , также поддерживают GNU Emacs. В таких системах, вероятно, полное руководство по make можно обнаружить в

info -системе документации Emacs.

На сайте FSF доступны версии

GNU make для DOS и Windows.

15.4.1. Базовая теория

make

При разработке программ на языках C или C++ важной частью для построения приложения является семейство команд компиляции и компоновки, необходимых для получения из

файлов исходного кода работающих бинарных файлов. Ввод данных команд — длительная и кропотливая работа, и большинство современных сред разработки включают в себя способ помещения их в командные файлы или базы данных, которые можно автоматически вызывать для сборки приложения.

Unix-программа

make(1) , родоначальник всех этих средств, была разработана специально для того, чтобы помочь С-программистам управлять данными инструкциями. Она позволяет описать зависимости между файлами проекта в одном или нескольких "make-файлах". Каждый make-файл состоит из последовательности

правил, каждое из которых указывает утилите make, что некоторый заданный целевой файл зависит от некоторого набора исходных файлов и определяет действия в случае, если любой из файлов исходного кода является более новым, чем целевой файл. Фактически программисту не требуется описывать все зависимости, поскольку программа

make способна установить большинство очевидных зависимостей по именам файлов и расширениям.

Например, программист может указать в make-файле, что бинарный файл myprog зависит от трех объектных файлов myprog.o, helper.o и stuff.o. Если имеются файлы исходного кода myprog.c, helper.c и stuff.c, то утилита

make без специальных указаний определит, что каждый .o-файл зависит от соответствующего .c-файла, и предоставит собственную стандартную инструкцию для сборки .o-файла из .c-файла.

Возникновение make связано с визитом ко мне Стива Джонсона (Steve Johnson — автор

уасс и других программ). Когда он пришел, он был очень недоволен тем, что ему пришлось потратить впустую утро, занимаясь отладкой корректной программы (ошибка была устранена, файл не был откомпилирован, и, следовательно, сс *.o не работала). А поскольку я потратил часть предыдущего вечера, справляясь с той же проблемой в разрабатываемом мною проекте, у нас появилась идея создания инструмента для решения данной задачи. Все началось с тщательно продуманной идеи анализатора зависимостей, потом свелось к нечто более простому и в те же выходные превратилось в

make . Использование инструментов, которые все еще оставались сырыми, было частью культуры. Make-файлы были текстовыми, а не "волшебно" закодированными бинарными файлами, поскольку это было в духе Unix: печатаемый, отлаживаемый, понятный материал.Стюарт Фельдман.

После ввода команды make, в каталоге проекта программа

make просматривает все правила и временные метки, после чего выполняет минимальный объем работы, необходимый для того, чтобы гарантировать актуальность производных файлов.

Хороший пример make-файла умеренной сложности можно взять из исходных кодов программы

fetchmail . В дальнейших подразделах он будет рассматриваться снова.

Очень сложные make-файлы (особенно, когда они вызывают вспомогательные make-файлы) могут стать источником осложнений вместо того, чтобы упростить процесс сборки. Ставшее классическим предупреждение впервые прозвучало в статье

"Recursive Make Considered Harmful" [131]. Аргумент в данной статье со времени ее публикации в 1997 году стал общепринятым и почти стал переворотом предыдущей практики в сообществе.

Обсуждение утилиты

make(1) будет неполным без признания того факта, что она включает в себя одну из худших недоработок конструкции в истории Unix. Использование символов табуляции в качестве необходимых начальных символов командных строк, связанных с правилом, означает, что интерпретация make-файла может радикально измениться из-за невидимых различий в пустых пространствах.

Почему в столбце 1 используется табуляция? Yacc была новой программой, а Lex ещё более новой. Я их еще не попробовал, поэтому предположил, что это было бы хорошим поводом для обучения. После того как я запутался, впервые попробовав Lex, я просто сделал нечто простое с моделью "конец строки-табуляция". Конструкция работала, и поэтому осталась без изменений. А спустя несколько недель сформировалось сообщество пользователей (около десятка человек), причем большинство из них были моими друзьями, и я не хотел им навредить. Остальное, к сожалению, уже стало историей. Стюарт Фельдман.

15.4.2. Утилита

make в разработке не на C/C++

Программа

make может оказаться полезной не только для программ на C/C++. Языки сценариев, подобные описанным в главе 14, могут не требовать традиционных этапов компиляции и компоновки, однако часто существуют другие виды зависимостей, с которыми поможет справиться утилита

make(1) .

Предположим, например, что часть кода фактически генерируется из файла спецификации с помощью одной из методик, описанных в главе 9. В данном случае программу

make можно использовать для связи файла спецификации и сгенерированного исходного кода. Такой подход гарантирует, что всякий раз при изменении спецификации и повторном выполнении make сгенерированный код будет автоматически обновлен.

Весьма распространенной является практика использования правил make-файлов в целях выражения инструкций для создания документации, так же как и кода. Часто такой подход используется для автоматического создания PostScript или другой производной документации из главных документов, написанных на каком-либо языке разметки (например, HTML или одном из языков создания документов в Unix, которые рассматриваются в главе 18). Фактически такое использование настолько широко распространено, что его стоит проиллюстрировать учебным примером.

15.4.2.1. Учебный пример: использование

make для преобразования файла документации

В make-файле программы

fetchmail, например, есть три правила, которые связывают файлы FAQ, FEATURES и NOTES с исходными HTML-файлами fetchmail-FAQ.html, fetchmail-features.html и design-notes.html.

HTML-файлы предназначены для просмотра на Web-странице программы fetchmail, но если Web-браузер не используется, то HTML-разметка делает эти файлы неудобными для просмотра. Поэтому FAQ, FEATURES и NOTES представляют собой простые текстовые файлы, предназначенные для быстрого просмотра с помощью редактора или программы-пейджера при чтении собственно исходного кода

fetchmail (или, возможно, для размещения на FTP-сайтах, не поддерживающих Web-доступ).

Простые текстовые формы могут быть получены из главных HTML-файлов с помощью распространенной программы

lynx(1) с открытым исходным кодом.

lynx — является Web-браузером для текстовых дисплеев. Однако если данную программу вызвать с параметром -dump, то она будет достаточно корректно функционировать в качестве преобразователя HTML-ASCII.

Правила make позволяют разработчику редактировать главные HTML-документы, не заботясь впоследствии о повторной ручной сборке простых текстовых форм, поскольку файлы FAQ, FEATURES и NOTES будут соответствующим образом при необходимости каждый раз создаваться заново.

15.4.3. Правила make

Некоторые из наиболее интенсивно используемых правил в типичных make-файлах вообще не выражают зависимостей. Они позволяют связать небольшие процедуры, которые разработчик хочет механизировать, например, создание дистрибутивного пакета или удаление всех объектных файлов для компиляции проекта с нуля.

Нефайловые правила были созданы -умышленно и существовали с первого дня. Правила "make all" и "clean" были моими собственными ранними соглашениями. Стюарт Фельдман.

Существует хорошо развитый набор соглашений о том, какие правила должны присутствовать и как они должны быть названы. Придерживаясь данных соглашений, разработчик создает более понятные и простые в использовании make-файлы.

all

Правило

all должно создавать все выполняемые файлы проекта. Обычно в

all нет явного критерия. Вместо этого правило ссылается на все цели верхнего уровня в проекте (и, вовсе неслучайно, документирует, каковы они). Традиционно

all должно быть первым правилом make-файла, так чтобы оно было единственным правилом, которое выполняется, когда разработчик вводит команду make без аргументов.

test

Запуск автоматизированного тестового пакета для программы, обычно состоящего из набора блочных тестов (unit tests)[132] для поиска регрессий, ошибок или других отклонений от ожидаемого поведения во время процесса разработки. Правило "test" также могут использовать конечные пользователи программы, для того чтобы убедиться, что их инсталляция функционирует корректно.

clean

Удаление всех файлов (таких как бинарные исполняемые и объектные файлы), которые обычно создаются во время выполнения команды make all. Команда make clean должна вернуть процесс сборки программного обеспечения в исходное состояние.

dist

Создание архива исходного кода (обычно с помощью программы

tar(1)), который можно распространять как единое целое и использовать для сборки программы заново на другой машине. Целью данной директивы является создание эквивалентного кода, зависящего от all таким образом, чтобы правило make dist автоматически заново собирало целый проект, прежде чем создать его дистрибутивный архив. Это хороший способ избежать ошибок, в результате которых в дистрибутив не включаются действительно необходимые производные файлы (например, простой текстовый файл README в

fetchmail , который фактически генерируется из HTML-файла).

distclean

Отбрасывает все, кроме того, что разработчик включил бы в случае упаковки исходного кода с помощью команды make dist. Действие может быть аналогичным команде make clean, но

distclean следует в любом случае включать как отдельное правило для документирования происходящего. Если действие отличается, то обычно оно отличается отбрасыванием локальных конфигурационных файлов, которые не являются частью обычной последовательности сборки make all (такой как последовательность, сгенерированная утилитой

autoconf(1)\autoconf(1) рассматривается в главе 17).

realclean

Отбрасывает все, что может быть заново собрано с помощью данного make-файла. Действие может быть таким же, как make distclean, но

realclean следует в любом случае включать как отдельное правило для документирования происходящего. Если действие отличается, то обычно оно отличается отбрасыванием файлов, которые являются производными, но (по какой-либо причине), так или иначе поставляются с исходными кодами проекта.

install

Инсталляция исполняемых файлов проекта и документации в системные каталоги таким образом, чтобы они были доступны пользователям (обычно данная операция требует полномочий администратора). Инициализация или обновление баз данных или библиотек, которые необходимы исполняемым файлам для работы.

uninstall

Удаление файлов, установленных в системные каталоги командой make install (обычно данная операция требует полномочий администратора). Эта операция должна быть полностью противоположной make install. Это правило означает подчинение соглашениям, которые ищут опытные пользователи Unix, так как для них они являются подтверждением продуманной конструкции. Напротив, его отсутствие является в лучшем случае небрежностью и (например, когда в ходе инсталляции создаются большие файлы базы данных) может рассматриваться как некомпетентность и невнимательность.

Работающие примеры всех стандартных целей доступны для изучения в make-файле программы

fetchmail . Их изучение позволит понять модель и полнее изучить структуру пакета

fetchmail . Одним из преимуществ использования данных стандартных правил является то, что они формируют полную схему проекта.

Однако для разработчика нет необходимости ограничивать себя данными правилами. Однажды научившись использовать

make , разработчик обнаруживает, что он все чаще использует механизм make-файлов для автоматизации небольших задач, которые зависят от состояния файлов проекта. Make-файл проекта — удобная центральная точка для организации данных задач. Его использование делает их легкодоступными для изучения и позволяет избежать загромождения рабочей области проекта небольшими случайными сценариями.

15.4.4. Генерация make-файлов

Одним из неочевидных преимуществ Unix

make по сравнению с базами данных зависимостей, встроенных в многие IDE-среды, является то, что make-файлы представляют собой простые текстовые файлы, т.е. файлы, которые могут создаваться программами.

В середине 1980-х годов в дистрибутивах крупных Unix-программ были достаточно распространены сложные специальные shell-сценарии, которые исследовали окружение и использовали собранную информацию для создания нестандартных make-файлов. Такие специальные конфигураторы достигали абсурдных размеров. Автор данной книги однажды написал такой конфигуратор, состоящий из 3000 строк shell-кода, почти вдвое больше любого отдельного модуля программы, для которой он был предназначен, и это не было необычным.

Однажды было решено положить этому конец, и многие представители сообщества настроились на написание инструментов, которые автоматизировали бы часть или весь процесс сопровождения make-файлов. Данные инструментальные средства обычно были призваны разрешить две проблемы.

Одной из проблем была

переносимость на другие платформы. Генераторы make-файлов в большинстве случаев создаются для работы на множестве различных аппаратных платформ и вариантов Unix. Как правило, они пытаются определить параметры локальной системы (включая все от размера машинного слова до инструментов, языков служебных библиотек и даже имеющихся в

системе программ форматирования документов). Затем генераторы пытаются использовать полученные сведения для написания make-файлов, которые используют средства локальной системы и компенсируют ее индивидуальные особенности.

Другой проблемой является

вывод зависимостей. Существует возможность получить множество сведений о зависимостях в семействе файлов исходного C-кода путем анализа самих файлов (особенно их директив #include). Многие генераторы make-файлов выполняют данные действия для автоматического создания make-зависимостей.

Способы достижения данных целей у всех генераторов make-файлов несколько различаются. Вероятно, использовалось не менее десяти генераторов, но большинство из них оказались неадекватными или слишком сложными в управлении или имели оба этих недостатка. Только несколько конфигураторов до сих пор активно используются. Ниже рассматриваются основные представители этого семейства программ. Все они доступны в Internet в виде программ с открытым исходным кодом.

15.4.4.1.

makedepend

Несколько небольших инструментов решают исключительно часть описанной выше проблемы, связанную с автоматизацией правил. Утилита

makedepend , распространяемая наряду с системой X Window разработки MIT, является самым быстрым и наиболее полезным из таких инструментов и предустанавливается на все современные Unix-системы, включая все дистрибутивы Linux.

makedepend принимает коллекцию исходных кодов C и генерирует зависимости для соответствующих .o-файлов из их директив #include. Их можно добавлять непосредственно в make-файл, и makedepend фактически предназначена именно для этого.

makedepend бесполезна для проектов, написанных не на C. Утилита не пытается решать несколько частей проблемы создания make-файлов. Однако то, что она делает, она делает особенно хорошо.

Утилита

makedepend полностью документирована на соответствующей странице руководства. Команда man makedepend, введенная в терминальном окне, быстро предоставит сведения, необходимые для запуска данной утилиты.

15.4.4.2.

Imake

Утилита

Imake была написана в попытке автоматизировать создание make-файлов для системы X

Window. Она надстраивается на

makedepend для решения как проблемы вывода зависимостей, так и проблемы переносимости.

Imake- система эффективно заменяет традиционные make-файлы Imake-файлами, написанными в более компактной и мощной форме, которая (эффективно) компилируется в make-файлы. В процессе компиляции используется файл правил, который считается специфическим для системы и включает в себя множество сведений о локальной среде.

Imake хорошо подходит для разрешения трудностей переносимости и конфигурации, специфических для X, и используется во всех проектах, которые являются частью дистрибутива X Window. Однако за пределами X-сообщества данная утилита не приобрела большой популярности. Она трудна для изучения, использования и расширения, а кроме того, генерирует make-файлы огромных размеров и сложности.

Imake -инструменты доступны на любой Unix-системе, поддерживающей X, включая Linux. Существует один "героический" проект [16], цель которого заключается в "прояснении тайн"

Imake для не X-программистов. Вопросы, освещаемые данным проектом, стоит изучить всем, кто собирается заниматься X-программированием.

15.4.4.3.

autoconf

Утилита

autoconf была написана программистами, которые изучили и отклонили подход

Imake . Утилита

autoconf генерирует для каждого проекта shell-сценарии configure, которые подобны старомодным специальным конфигураторам, configure-сценарии способны генерировать make-файлы (в том числе).

Утилита

autoconf направлена на разрешение проблемы переносимости и вообще не выполняет встроенного вывода зависимостей. Несмотря на то, что данная программа, вероятно, такая же сложная, как

Imake, она является гораздо более гибкой и проще расширяется. Вместо использования базы данных правил в системе, утилита генерирует shell-код configure, который осматривает систему, определяя необходимые параметры.

Каждый configure-сценарий создается на основе уникального для проекта шаблона configure.in, который должен написать разработчик. Однажды сгенерированный сценарий configure является самодостаточным и способен конфигурировать данный проект на системах, которые не содержат саму утилиту

autoconf(1) .

Подход к созданию make-файлов, принятый для

autoconf , подобен подходу Imake в том, что разработчик начинает с написания шаблона make-файла для своего проекта. Однако файлы Makefile.in, созданные

autoconf , по существу являются просто make-файлами с метками-заполнителями для простой текстовой замены; не существует второй формы записи, которую требуется изучать. Если требуется осуществить вывод зависимостей, то необходимо предпринять явные шаги для вызова

makedepend(1) или подобного инструмента, или использовать утилиту

automake(1) .

Утилита

autoconf документирована в руководстве в GNU

info -формате. Исходные сценарии autoconf доступны на сайте архива FSF, а кроме того, они предустановлены на многих Unix и Linux-системах. Упомянутое руководство можно просматривать с помощью справочной системы Emacs.

Несмотря на отсутствие непосредственной поддержки вывода зависимостей и характерного для

autoconf узкоспециального подхода, в середине 2003 года данная утилита, несомненно, была наиболее популярной из всех генераторов make-файлов. Она превзошла

Imake и стала причиной выхода из употребления, по крайней мере, одного главного конкурента (

metaconfig).

Существует справочник

"GNU Autoconf Automake and Libtool" [86]. Дополнительная информация по утилите

autoconf в несколько ином аспекте рассматривается в главе 17.

15.4.4.4.

automake

Утилита

automake — это попытка добавить Imake-подобную функцию вывода зависимостей как уровень над

autoconf(1). Разработчик пишет шаблоны Makefile.am в форме записи, которая явно подобна

Imake- нотации. Затем утилита

automake(1) компилирует их в файлы Makefile.in, которыми впоследствии оперируют

autoconf- сценарии configure.

automake в середине 2003 года все еще оставалась сравнительно новой технологией. Она

используется в нескольких FSF-проектах, но еще не принята широко за пределами FSF-сообщества. Несмотря на то, что ее общий подход выглядит многообещающе, утилита все еще является довольно хрупкой — она работает в стереотипных условиях, но склонна к серьезным сбоям, если попытаться использовать ее необычным путем.

Полная документация поставляется с утилитой

automake , которую можно загрузить с сайта архива FSF.

15.5. Системы контроля версий

Как известно, по мере того как проект движется от первого прототипа к распространяемой версии, код проходит через несколько циклов развития, в ходе которых разработчик исследует новые области, отлаживает, а затем стабилизирует достижения.

И такое развитие не прекращается с выходом первой версии продукта. Большинство проектов нуждаются в сопровождении и усовершенствовании после стадии 1.0, и уже впоследствии появляется множество версий. Отслеживание всех деталей этого процесса является той задачей, с которой компьютеры справляются лучше человека.

15.5.1. Для чего используется контроль версий

Развитие кода поднимает несколько практических проблем, которые могут быть основными причинами противоречий и монотонной работы, а следовательно, и серьезного снижения продуктивности. Время, потраченное на разрешение данных проблем, — это время, не уделенное правильной разработке конструкции и функциональности проекта.

Вероятно, наиболее важной проблемой является

обратное восстановление . Если разработчик вносит изменение и выясняет, что оно нежизнеспособно, то каким образом можно вернуться к заведомо хорошей версии кода? Если процесс восстановления труден и ненадежен, то разработчику вообще тяжело решиться на внесение изменений (можно испортить весь проект или потратить очень много времени).

Почти настолько же важным является

отслеживание изменений . Известно, что код изменился, но известно ли, почему именно? Вполне возможно, что причины изменений вскоре будут забыты, а позднее возникнут снова. Если в проекте участвуют другие разработчики, то каким образом узнать, что именно они изменили и кто ответственен за каждое изменение?

Очень полезно спрашивать себя, что

изменилось с момента последней заведомо хорошей версии, даже если в проекте не участвуют другие разработчики. Часто это позволяет обнаружить нежелательные изменения, такие как забытый отладочный код. Сейчас я делаю это регулярно перед регистрацией группы изменений.Генри Спенсер.

Еще одним вопросом является

отслеживание ошибок . Очень часто поступают отчеты об ошибках для определенной версии после того, как код значительно изменился по сравнению с этой версией. Иногда разработчик может немедленно определить, что ошибка уже исправлена, но часто это невозможно. Предположим, она не воспроизводится в новой версии. Как вернуть код к состоянию старой версии, для того чтобы воспроизвести ошибку и разобраться с ней?

Чтобы решить данную проблему, необходимы процедуры для сохранения истории проекта и снабжения его комментариями, объясняющими историю. Если в проекте участвует несколько разработчиков, то также понадобятся механизмы, которые позволяют быть уверенным, что разработчики не изменяют чужие версии.

15.5.2. Контроль версий вручную

Самым примитивным (но все еще очень распространенным) является ручной метод. Разработчик периодически делает снимки проекта, создавая его резервные копии, включает исторические комментарии в файлы исходного кода, а также устно или по электронной почте договаривается с другими разработчиками не изменять определенные файлы, пока он над ними работает.

Скрытые затраты такого метода высоки, особенно когда (как часто случается) происходит сбой. Такие процедуры требуют времени и концентрации. Они чреваты ошибками и склонны "ускользать", когда проект сталкивается с трудностями, т.е. именно тогда, когда они больше всего нужны.

Как и большинство ручной работы, данный метод невозможно хорошо масштабировать. Он ограничивает детализацию отслеживания изменений и склонен к потере таких подробностей, как порядок изменений, имен их авторов и указания причин. Восстановление только части крупного изменения может оказаться утомительной и отнимающей много времени процедурой, и часто разработчики вынуждены восстанавливать более раннюю версию.

15.5.3 Автоматизированный контроль версий

Для того чтобы избежать описанных выше проблем, можно использовать какую- либо систему контроля версий (Version-Control System — VCS), пакет программ, который автоматизирует большую часть рутинной работы по поддержанию аннотированной истории проекта и позволяет избежать конфликтов модификации.

Большинство VCS-систем используют одну и ту же базовую логику. Использование такой системы начинается с

регистрации семейства файлов исходного кода, т.е. с указания VCS-системе начать архивирование файлов, описывая историю их изменения. После этого при необходимости отредактировать один из таких файлов требуется

отметить (check out) данный файл — объявить его исключительную блокировку. По окончании редактирования необходимо

сдать (check in) файл, добавляя внесенные изменения в архив, снимая блокировку и вводя комментарии, поясняющие суть внесенных изменений.

История проекта не обязательно имеет линейный характер. Все широко используемые VCS-системы фактически позволяют разработчику поддерживать дерево вариантных версий (например, версии для других машин) с помощью инструментов для объединения ветвей обратно в главную "стволовую" версию. Данная функция становится важной по мере увеличения размера и дисперсии группы разработки. Однако ее следует использовать с осторожностью. Множество активных вариантов кодовой базы могут создавать путаницу (только связанные с правильной версией отчеты об ошибках не всегда оказываются простыми), и автоматизированное слияние ветвей не гарантирует, что комбинированный код будет работать.

Остальное в работе VCS в основном направлено на удобство использования: функции маркирования и отчета, окружающие данные базовые операции, инструменты, позволяющие разработчику просматривать отличия между версиями или группировать заданный набор версий файлов как именованную

редакцию , которую можно изучать или к которой можно в любое время вернуться без потери более поздних изменений.

VCS-системы имеют собственные проблемы. Наибольшей из них является то, что использование VCS предполагает дополнительные шаги каждый раз, когда необходимо отредактировать какой-либо файл, шаги, которые разработчики склонны пропускать в спешке, если их приходится делать вручную. В конце данной главы обсуждается способ разрешения данной проблемы.

Другая проблема состоит в том, что некоторые виды естественных операций часто дезориентируют VCS-системы. Печально известная слабость VCS-систем — переименование файлов. Не просто автоматически гарантировать, что история изменений файла будет сопровождать его после переименования. Проблемы переименования особенно трудно разрешить, когда VCS-система поддерживает ветвление.

Вопреки данным трудностям, VCS-системы во многих смыслах являются большим благом, позволяющим добиться высокой продуктивности и качества кода, даже для небольших проектов с одним разработчиком. Они автоматизируют многие процедуры, которые являются только утомительной работой. Они серьезно помогают в восстановлении после ошибок. Возможно, наиболее важным является то, что они дают программистам свободу для экспериментов, гарантируя, что восстановление заведомо исправного состояния всегда будет простым.

Кроме того, системы VCS полезны не только для программного кода. Рукопись данной книги во время написания поддерживалась как совокупность файлов в системе

RCS .

15.5.4. Unix-инструменты для контроля версий

Историческое значение в мире Unix имеют три VCS-системы; они рассматриваются в данном разделе. Более развернутое введение и учебные материалы приведены в книге

"Applying RCS and SCCS" [5].

15.5.4.1. Source Code Control System (SCCS)

Первой из рассматриваемых систем появилась

SCCS , оригинальная система управления исходным кодом (Source Code Control System), разработанная в Bell Labs примерно в 1980 году и представленная в System III Unix.

SCCS — это, вероятно, первая серьезная попытка создания унифицированной системы управления исходным кодом. Передовые идеи, впервые реализованные в ней, до сих пор встречаются на некотором уровне во всех последующих системах, включая коммерческие Unix- и Windows-продукты, такие как ClearCase.

Однако сама

SCCS в настоящее время устарела. Она была частной собственностью Bell Labs. С тех пор были разработаны превосходные альтернативы с открытыми исходными кодами, и большая часть Unix-сообщества перешла к их использованию.

SCCS до сих пор используется некоторыми коммерческими поставщиками программного обеспечения для управления старыми проектами, однако для новых проектов ее рекомендовать нельзя.

Не существует ни одной реализации

SCCS с полностью открытым исходным кодом. Клон, который называется CSSC (Compatibly Stupid Source Control), разрабатывается при поддержке FSF.

15.5.4.2. Revision Control System (RCS)

Список превосходных альтернатив с открытым исходным кодом начинается с системы RCS (Revision Control System — система управления ревизиями), которая была создана в Университете Пурдью через несколько лет после

SCCS и первоначально распространялась с 4.3BSD Unix. Данная система логически подобна

SCCS, но имеет более четкий командный интерфейс и хорошие средства для группировки целых редакций проекта под символическими именами.

RCS в настоящее время является наиболее широко используемой системой в мире Unix. В некоторых других Unix-системах контроля версий RCS используется в качестве серверной части или базового уровня. Она хорошо подходит для проектов, разрабатываемых одним разработчиком или небольшой группой разработчиков, находящихся в одной лаборатории.

Исходные коды

RCS курируются и распространяются FSF. Существуют бесплатные версии для операционных систем Microsoft и VAX VMS.

15.5.4.3. Concurrent Version System (CVS)

CVS (Concurrent Version System — система параллельных версий) была разработана в начале 1990-х годов как клиентская часть к

RCS , но модель контроля версий, которая использовалась в ней, настолько отличалась, что данная система была немедленно квалифицирована как новая конструкция. Современные реализации не основываются на

RCS .

В отличие от

RCS и

SCCS , система

CVS не выполняет исключительную блокировку файлов, когда они отмечаются разработчиком для редактирования. Вместо этого система пытается автоматически согласовать не противоречащие друг другу изменения во время сдачи файлов, а в случае возникновения конфликтов требует вмешательства пользователя. Такая конструкция работает, поскольку конфликты заплат гораздо менее распространены, чем можно было бы предполагать.

Интерфейс

CVS значительно более сложен, чем интерфейс

RCS , и требует большего дискового пространства. Данные свойства делают систему неудачным выбором для небольших проектов. С другой стороны,

CVS хорошо подходит для крупных проектов с участием множества разработчиков, распространенных по нескольким участкам разработки, которые связаны друг с другом посредством Internet. CVS-инструменты на клиентской машине можно легко настроить на непосредственное взаимодействие с репозиторием, расположенным на другом узле.

В сообществе открытого исходного кода система

CVS интенсивно используется для таких проектов, как GNOME и Mozilla. Обычно такие CVS-репозитории позволяют любому разработчику отмечать исходные файлы удаленно. Следовательно, кто угодно может создать локальную копию проекта, модифицировать ее и отправить заплаты с изменениями по почте кураторам проекта. Реальный доступ на запись к такому репозиторию более ограничен и должен быть явно предоставлен кураторами проекта. Разработчик, имеющий такой доступ, может обновить проект из своей модифицированной локальной копии, в результате чего локальные изменения вносятся непосредственно в удаленный репозиторий.

Пример хорошо организованного CVS-репозитория, доступного через Internet, можно посмотреть на сайте GNOME CVS <http://cvs.gnome.org>. На сайте иллюстрируется использование инструментов быстрого просмотра с поддержкой

CVS , таких как Bonsai, которые полезны тем, что способствуют координации работы крупной и децентрализованной группы разработчиков.

Общественная структура и философия, сопровождающие использование

CVS , являются настолько же важными, насколько подробности использования инструментов. Предполагается, что проект

будет открытым и децентрализованным, а код будет предметом экспертной оценки и изучения даже со стороны разработчиков, которые официально не являются членами проектной группы.

Не менее важна "неблокирующая" философия

CVS , которая означает, что проект не может быть заблокирован в случае, если программист покинет его на полпути, не разблокировав модифицируемый им файл. Таким образом,

CVS позволяет разработчикам избегать "единоличной точки сбоя". В свою очередь, это означает, что границы проекта могут быть "плавающими", эпизодическое участие в проекте является сравнительно простым, и для проектов не требуется сложная управленческая иерархия.

Исходные коды системы

CVS сопровождаются и распространяются FSF.

CVS имеет значительные проблемы. Некоторые из них являются просто ошибками реализации, однако основная проблема заключается в том, что пространство имен файлов проекта не контролируется тем же способом, что и изменения в файлах. Поэтому

CVS легко запутать переименованием, удалением и добавлением файлов. Кроме того, CVS регистрирует изменения для каждого файла, а не для

групп изменений файлов. Это усложняет возврат к определенным версиям и обработку частичного возвращения файлов. Вместе с тем, ни одна из указанных проблем не является собственной для неблокирующего стиля, и такие проблемы успешно решаются более новыми системами контроля версий.

15.5.4.4. Другие системы контроля версий

Конструктивные проблемы системы

CVS достаточны для того, чтобы создать потребность в лучших VCS-системах с открытым исходным кодом. Несколько таких проектов полным ходом разрабатывались в 2003 году. Наиболее выдающимися из них являются проекты

Aegis и

Subversion .

Проект Aegis <http://www.pcug.org.au/~millerp/aegis/aegis.html> имеет самую длинную историю среди CVS-альтернатив и является зрелой действующей системой. С 1991 года данная система обеспечивает контроль версий для самого проекта Aegis. Особый акцент в этой системе сделан на возвратное тестирование и проверку достоверности.

Система Subversion <http://subversion.tigris.org/> позиционируется как "правильно сделанная CVS", с полностью разрешенными известными конструктивными проблемами. В 2003 году данная система имела наилучшие перспективы заменить в ближайшем будущем

CVS .

Проект BitKeeper <http://www.bitkeeper.com> исследует некоторые интересные

конструктивные идеи, связанные с группами изменений и множественными распределенными репозиториями кода. Линус Торвальдс использует Bitkeeper для исходных кодов ядра Linux. Однако лицензия (отличная от лицензий на открытый исходный код) на данную систему является спорной и значительно задерживает признание данного продукта сообществом.

15.6. Отладка времени выполнения

Каждый, кто занимается программированием больше одной недели, знает, что исправление синтаксических ошибок является

простой частью отладки. За ней следует сложная часть, когда необходимо разобраться, почему поведение синтаксически корректной программы не соответствует ожидаемому.

Традиции Unix побуждают разработчиков предупреждать эту проблему путем проектирования прозрачных конструкций — в частности, путем проектирования программ таким образом, чтобы можно было без труда невооруженным глазом и с помощью простых инструментов осуществлять мониторинг внутренних потоков данных в программах, а также просто создавать их ментальные модели. Данная тема подробно рассматривалась в главе 6. Создание прозрачных конструкций важно как для предотвращения ошибок, так и для упрощения задач отладки времени выполнения.

Однако одного только проектирования прозрачных конструкций недостаточно. При отладке программы во время выполнения чрезвычайно полезно иметь возможность изучать состояние выполняемой программы, устанавливать точки останова и контролируемым способом выполнять блоки программы вплоть до уровня одного оператора. В операционной системе Unix имеется давняя традиция поддержки программ, способствующих решению данных проблем. В состав Unix-систем с открытыми исходными кодами входит одно мощное средство,

gdb (еще один FSF-проект), которое поддерживает отладку кода, написанного на C и C++.

Языки Perl, Python, Java и Emacs Lisp поддерживают стандартные пакеты или программы (которые включаются в состав их базовых дистрибутивов), позволяющие устанавливать контрольные точки, управлять выполнением и осуществлять общие операции отладки во время выполнения. Tcl, разработанный как небольшой язык для небольших проектов, не имеет такого средства (хотя он имеет средство трассировки, которое можно использовать для наблюдения за переменными во время выполнения).

Для разработчика важно правильно понимать философию Unix и тратить свое время не на низкоуровневые детали, а на качество конструкции, а также автоматизировать все, что возможно, включая кропотливую работу по отладке программы во время выполнения.

15.7. Профилирование

Общее правило: 90% времени выполнения программы тратится на 10% ее кода. Профайлеры представляют собой инструменты, способствующие идентификации этих 10% "горячих точек", которые ограничивают скорость программы, а значит, профайлеры — это хороший способ повышения скорости.

Однако в традиции Unix профайлерам отводится гораздо более важная функция. Они позволяют разработчику

не оптимизировать оставшиеся 90%. И это не только сокращает объем работ.

Действительно ценный эффект заключается в том, что программист, который не оптимизирует 90% кода, сдерживает глобальную сложность и сокращает количество ошибок.

В данной связи можно процитировать Дональда Кнута: "Преждевременная оптимизация — корень всех зол". Это голос опыта. Необходимо тщательно проектировать конструкцию и, прежде всего, подумать, что является

верным . Регулировку в целях повышения эффективности можно сделать позднее.

В этом разработчику помогают профайлеры. Если выработать полезную привычку использовать их, то можно избавиться от вредной привычки преждевременной оптимизации. Профайлеры изменяют не только способ работы программиста, но и образ его мышления.

Профайлеры для компилируемых языков зависят от измерения параметров объектного кода, поэтому они еще больше зависят от платформы, чем компиляторы. С другой стороны, профайлер компилируемого языка не заботится об исходном языке измеряемой им программы. В Unix один профайлер

gprof(1) обрабатывает C, C++ и все остальные компилируемые языки.

Языки Perl, Python и Emacs Lisp имеют собственные профайлеры, включенные в их базовые дистрибутивы. Такие профайлеры переносятся на все платформы, где работают данные языки. В языке Java имеется встроенное профилирование. Tcl все еще не имеет поддержки профилирования.

15.8. Комбинирование инструментов с Emacs

Одной из областей, где редактор Emacs весьма хорошо применим, является его использование в качестве интерфейсной части для других инструментов разработки (эта тема рассматривалась с философской точки зрения в главе 13). Действительно, почти всеми инструментами, рассмотренными в данной главе, можно управлять из сеанса редактора Emacs посредством интерфейсных частей, которые увеличивают эффективность данных инструментов по сравнению с их автономным использованием.

Для иллюстрации этого факта ниже рассматривается использование данных инструментов совместно с Emacs в обычном цикле компиляция/тестирование/отладка. Подробнее данная тема описана в собственной справочной системе Emacs. В этом разделе предоставляется общий обзор, который подтолкнет читателя к дальнейшему изучению.

Разработчику необходимо читать и учиться — не только использованию Emacs, но и выработке ментальной склонности к поиску синергии между программами и ее созданию. Данный раздел рекомендуется читать как инструкцию в философском смысле, а не только в методическом.

15.8.1. Emacs и

make

Например, утилиту

make можно запустить из Emacs с помощью команды ESC-x compile [Enter]. Данная команда запускает

make(1) в текущем каталоге, собирая вывод в буфер Emacs.

Сама по себе данная операция не была бы очень полезной, но Emacs-режим

make распознает формат сообщений об ошибках (указывая исходный файл и номер строки), которые генерируются Unix C-компиляторами и многими другими инструментами.

Если какая-либо выполняемая

make инструкция генерирует сообщения об ошибках, то команда Ctl-X ` (Ctrl-X-обратная кавычка) пытается выполнить их синтаксический анализ и последовательно переходит к каждой ошибке, открывая окно соответствующего файла и перемещая курсор к строке с ошибкой[133].

Данная возможность чрезвычайно упрощает просмотр всей сборки с исправлением синтаксиса, который был нарушен с момента последней компиляции.

15.8.2. Emacs и отладка во время выполнения

Для обнаружения ошибок времени выполнения Emacs предоставляет аналогичную возможность интеграции с символическим отладчиком, т.е. разработчик может использовать какой-либо Emacs-режим для установки контрольных точек в программах и изучения их состояния во время выполнения. Отладчик запускается путем передачи ему команд через окно Emacs. Каждый раз, когда отладчик останавливается в контрольной точке, сообщение об источнике ошибки, возвращаемое отладчиком, анализируется и используется во всплывающем окне в области, охватывающей контрольную точку исходного файла.

Emacs-режим Grand Unified Debugger (большой унифицированный отладчик) поддерживает все основные отладчики C:

gdb(1), sdb(1), dbx(1) и

xdb(1) . Он также поддерживает символический отладчик Perl с использованием модуля perldb и стандартные отладчики для Java и Python. Средства, встроенные в сам Emacs Lisp, поддерживают интерактивную отладку кода Emacs Lisp.

К моменту написания книги (середина 2003 года) еще не существовало поддержки для Tcl-отладки из Emacs. Конструкция Tcl такова, что вряд ли когда-либо такая поддержка будет добавлена.

15.8.3. Emacs и контроль версий

Сразу после исправления программного синтаксиса и устранения ошибок времени выполнения часто требуется сохранить внесенные изменения в архив системы контроля версий. Однако не многие разработчики хотят вводить команды входного и выходного контроля версий при каждой операции редактирования.

К счастью, Emacs может помочь и в данной ситуации. Код, встроенный в Emacs, реализует простой в использовании пользовательский интерфейс к системам

SCCS, RCS, CVS или Subversion. Команда Ctl-x v v пытается определить логически следующую операцию контроля версий, которую необходимо выполнить для редактируемого файла. В число данных операций входят регистрация файла, его отметка и блокировка, а также возвращение (комментарии по изменениям принимаются во всплывающем буфере) [134].

Кроме того, Emacs помогает просматривать историю изменений контролируемых файлов, а также отказаться от нежелательных изменений. Emacs упрощает применение операций контроля версий к целым группам файлов или деревьям каталогов файлов в проекте. В целом, Emacs выполняет значительную работу, делая операции контроля версий безболезненными.

Последствия использования данных функций гораздо серьезнее, чем это обычно предполагается. Разработчик, который привык к быстрому и простому контролю версий, вскоре обнаруживает, что это дает значительную свободу эксперимента. Зная, что всегда можно вернуться к заведомо исправному состоянию, программист чувствует себя свободнее в разработке гибким, исследовательским путем, испытывая множество изменений и изучая их влияние.

15.8.4. Emacs и профилирование

Возможно, единственной фазой цикла разработки, в которой интерфейсные возможности Emacs

не предоставляют реальной помощи, является профилирование. Профилирование, в сущности, является пакетной операцией — внедрение профайлера в программу, ее запуск, просмотр статистики, редактирование в целях повышения скорости, повторение процесса. В специфических для профилирования частях цикла разработки нет достаточного пространства для применения мощных средств Emacs.

Тем не менее, существует веская причина обдумать использование Emacs для профилирования. Если читателю приходится анализировать

большое количество отчетов профайлеров, то, возможно, стоит написать режим, в котором щелчок мыши или клавиатурная комбинация в строке отчета профайлера позволит просмотреть исходный код соответствующей функции. Фактически данную идею достаточно просто было бы реализовать, используя Emacs для "отметки" кода. В действительности, к моменту прочтения данной книги какой-либо другой читатель, может быть, уже написал такой режим и внес его в открытую базу кода Emacs.

Реальная цель в данном случае также является философской. Не следует выполнять монотонную работу — это пустая трата времени и продуктивности. Если выясняется, что разработчик тратит много времени на низкоуровневые механические части разработки, то следует вернуться назад, воспользоваться философией Unix, применить имеющийся

инструментарий для полной или частичной автоматизации задачи.

А затем вернуть часть наработок в качестве "платы" за все унаследованные знания путем публикации в Internet своего решения в качестве программного обеспечения с открытым исходным кодом. Помогите своим коллегам-программистам также освободиться от монотонной работы.

15.8.5. Лучше, чем IDE

Ранее в данной главе утверждалось, что Emacs способен предоставить программисту возможности, аналогичные возможностям какой-либо традиционной интегрированной среды разработки и даже превосходящие их. К настоящему моменту у читателя должно быть достаточно фактов, позволяющих подтвердить это. В Emacs можно разрабатывать целые проекты, управляя низкоуровневой механикой с помощью нескольких клавиатурных комбинаций и предохраняя себя от излишней нагрузки и необходимости постоянного переключения между контекстами.

Emacs-стиль разработки лишен некоторых возможностей развитых IDE-систем, например, графического изображения структуры программы. Но такие возможности, по существу, являются излишествами. Взамен Emacs предоставляет программисту гибкость и управление. Программист не ограничен рамками воображения разработчика IDE: используя Emacs Lisp, можно настраивать, подгонять под себя и добавлять в Emacs связанную с задачей логику. Кроме того, Emacs лучше, чем традиционные IDE-системы, проявляет себя в поддержке разработки на нескольких языках программирования.

Наконец, разработчик не ограничен тем, что одна небольшая группа разработчиков IDE-среды посчитала возможным поддерживать. Поддерживая связь с сообществом открытого исходного кода, можно воспользоваться результатами работы тысяч коллег, использующих Emacs разработчиков, которые сталкиваются с подобными трудностями. Это гораздо эффективнее и гораздо интереснее.

16

Повторное использование кода: не изобретая колесо

Когда великий человек воздерживается от действий, его сила чувствуется за тысячу миль.
—Тао Ти Чинг (популярный неправильный перевод)

Нежелание выполнять ненужную работу считается великой добродетелью у программистов. Если бы китайский мудрец Jlao-Цзы в наши дни все еще продолжал проповедовать учение Тао, то, возможно, его высказывание ошибочно переводили бы так: когда великий программист воздерживается от кодирования, то его сила чувствуется за тысячу миль. Действительно, современные переводчики предположили, что китайское понятие

ву-вей , которое обычно передавалось словами "бездействие" или "воздержание от действия", вероятно, должно читаться как "наименьшее действие" или "наиболее эффективное действие", или как "действие в соответствии с естественным правом", что даже

лучше описывает хорошую инженерную практику.

Следует помнить правило экономии. Повторные "поиски огня" и "изобретение колеса" для каждого нового проекта крайне расточительны. Время мышления дорого и весьма ценно по сравнению со всеми остальными производственными затратами при разработке программного обеспечения. Соответственно, его следует тратить на разрешение новых проблем, а не на переформулировку старых, для которых уже существуют известные решения. Такая позиция приносит наилучшую отдачу, как в понятиях интеллектуального капитала, так и в понятиях экономической эффективности инвестиций.

Изобретать заново колесо плохо не только потому, что при этом впустую тратится время, но и потому, что при этом часто создаются квадратные колеса. Существует почти непреодолимый соблазн сэкономить на времени переизобретения, используя сырую и слабо продуманную версию, а это в долгосрочной перспективе часто оказывается ложной экономией. Генри Спенсер.

Наиболее эффективный способ избежать изобретения колеса заключается в заимствовании имеющейся конструкции и реализации, иными словами, в повторном использовании кода.

Операционная система Unix поддерживает повторное использование кода на всех уровнях от отдельных библиотечных модулей до целых программ, которые система позволяет использовать в сценариях и соединять друг с другом. Систематическое использование имеющегося кода является одним из наиболее важных отличий поведения Unix-программистов, а опыт использования Unix, как правило, вырабатывает привычку пытаться создавать опытные решения путем комбинирования существующих компонентов с минимальным количеством новых изобретений, а не увлекаться написанием автономного кода, который будет использоваться только однажды.

Эффективность повторного использования кода является одной из великих непреходящих истин в разработке программного обеспечения. Однако многие разработчики, входящие в Unix-сообщество, имея базовый опыт в других операционных системах, не научились (или разучились) систематическому повторному использованию кода. Потери времени и двойная работа являются обычными, даже несмотря на то, что это противоречит интересам тех, кто платит за код, и тех, кто его производит. Осознав, почему такое неправильное поведение сохраняется, разработчик делает первый шаг к его изменению.

16.1. История случайного новичка

Почему программисты изобретают колесо? Существует множество причин от исключительно технических до психологических и причин связанных с экономикой систем производства программного обеспечения. Ущерб, наносимый распространенными затратами времени программистов, также затрагивает все эти уровни.

Рассмотрим, например, первый, формирующий рабочий опыт Дж. Рэндома Ньюби, программиста, недавно окончившего колледж. Предположим, что он глубоко осознал ценность повторного использования кода и переполнен юношеским рвением реализовать приобретенные знания на практике.

Во время работы над первым проектом Ньюби входит в состав коллектива, создающего крупное приложение. В качестве примера можно предположить, что это GUI-интерфейс, предназначенный для того, чтобы облегчить для конечных пользователей создание запросов и навигацию в крупной базе данных. Менеджеры проекта собрали подходящую, по их

мнению, коллекцию инструментов и компонентов, включая не только язык разработки, но также и многие библиотеки.

Библиотеки критически важны для данного проекта. Они включают в себя многие службы — от элементов управления окнами и сетевых соединений до целых систем, таких как интерактивная справка, которые в противном случае потребовали бы огромного количества дополнительного кода и ощутимо повлияли бы на бюджет проекта и дату его завершения.

Ньюби слегка озабочен датой завершения проекта. Ему может не хватать опыта, но он прочел

Dilbert и слышал несколько "боевых" историй от опытных программистов. Ньюби знает, что менеджеры имеют склонность к тому, что можно иносказательно назвать "агрессивным" расписанием. Возможно, он прочел

"Death March" Эда Йордона (Ed Yourdon) [91], который в 1996 году отмечал, что большинство проектов слишком, по крайней мере, на 50% лимитированы в отношении времени выполнения и ресурсов, и что этот лимит усугубляется.

Однако Ньюби умен и энергичен. Он полагает, что для него наилучшая возможность преуспеть заключается в как можно более разумном изучении методики использования переданных ему инструментов и библиотек. Он разминает пальцы, устремляется навстречу трудностям... и попадает в ад.

Все длится дольше и болезненнее, чем он предполагал. Под глянцевой поверхностью демонстрационных приложений повторно используемые Новичком компоненты, кажется, имеют крайние случаи, когда они ведут себя непредсказуемо или пагубно, — крайние случаи, с которыми его код сталкивается ежедневно. Ньюби пытается понять, о чем думали программисты библиотек, но не может, потому что компоненты недостаточно документированы, притом зачастую "техническими писателями", которые не являются программистами и думают не так, как программисты. Кроме того, Ньюби не может читать исходный код, для того чтобы понять, какие функции он фактически выполняет, поскольку библиотеки представляют собой малопонятные блоки объектного кода под коммерческими лицензиями.

Ньюби вынужден создавать все более сложные обходные пути для решения проблем компонентов до той точки, где чистый выигрыш от использования библиотек начинает выглядеть незначительным. Обходные пути постепенно делают его код неряшливым. Он, вероятно, сталкивается с несколькими ситуациями, в которых библиотеку просто невозможно заставить выполнять какую-либо критически важную задачу, теоретически входящую в спецификации библиотеки. Иногда он уверен, что существует некоторый способ действительно заставить "черный ящик" работать, но не может постичь его.

Ньюби находит, что по мере приложения все больших усилий к библиотекам время отладки экспоненциально растет. Его код запутан аварийными отказами и утечками памяти, следы которых ведут к библиотекам, в код, который Ньюби не может просмотреть или модифицировать. Он знает, что большинство этих следов, вероятно, ведут обратно к его коду, но без исходного текста их очень сложно отследить через код, которого он не писал.

Ньюби все больше расстраивается. В колледже он слышал, что производительность, равная сотне строк завершенного кода в неделю, считается хорошей. Тогда он смеялся, поскольку работал в несколько раз продуктивнее, выполняя лабораторные проекты и создавая программы ради удовольствия. Теперь удовольствия больше нет. Он борется не только со своей неопытностью, но и с каскадом проблем, созданных безответственностью или некомпетентностью других, — проблем, которые он может только обойти, а не устранить.

Расписание проекта нарушается. Ньюби, который мечтал быть архитектором программ, чувствует себя каменщиком, пытающимся построить из кирпичей то, что по существу не складывается и рассыпается под давлением нагрузки. Но его руководители не хотят слышать оправданий неопытного программиста. Слишком громкие жалобы на низкое качество компонентов, вероятнее всего, приведут его к "политическим" неприятностям с вышестоящими руководителями и менеджерами, выбравшими их. Но даже если он сумеет выиграть эту битву, изменение компонентов было бы сложным предложением, которое потребовало бы услуг адвокатов, специализирующихся на лицензионных понятиях.

Если Ньюби не будет очень и очень удачлив, то ему не удастся устранить ошибки библиотек в течение проекта. Рассуждая здраво, он может осознавать, что работающий код в библиотеках не привлекает его внимание так, как ошибки и упущения. Для прояснения ситуации он хотел бы поговорить с разработчиками компонентов. Ньюби полагает, что это разумные люди, подобные ему программисты, просто они работают в системе, которая подавляет их попытки делать правильные вещи. Однако он даже не может выяснить, кто они, а если бы мог, то поставщики программного обеспечения, на которых они работают, вероятно, не позволили бы им общаться с Ньюби.

В отчаянии Ньюби начинает создавать собственные "кирпичи" — моделируя менее стабильные библиотечные службы по образу более стабильных и создавая собственные реализации с нуля. Созданный им замещающий код, ввиду того, что Ньюби имеет его полную ментальную модель, которую может освежить в памяти путем повторного чтения, часто работает сравнительно хорошо, и его проще отлаживать, чем комбинацию малопонятных компонентов и обходных путей, которые он заменяет.

Ньюби получает урок; чем меньше он полагается на чужой код, тем больше работающих строк он может написать. Этот урок питает его эго. Как все молодые программисты, Ньюби считает себя умнее всех остальных. Кажется, его опыт поверхностно подтверждает это. Он начинает создавать свой собственный личный инструментарий, который лучше приспособлен к его потребностям.

К сожалению, эгоцентричные рефлексы, которые приобретает Ньюби, — краткосрочная локальная оптимизация, вызывающая долгосрочные проблемы. Он может написать больше строк кода, но фактическая ценность того, что он создает, вероятно, в значительной степени снижается по сравнению с тем, что было бы, если бы Ньюби добился успеха в повторном использовании кода. Больший код — отнюдь не лучший код. Код также не становится лучше, когда он написан на более низком уровне, где почти наверняка "изобретается колесо".

При смене работы Ньюби имеет в запасе, по крайней мере, еще один деморализующий опыт. Он, вероятно, обнаружит, что не может взять с собой свой инструментарий. Если он покинет здание с кодом, который написал в рабочее время, его прежние работодатели вполне могут рассматривать это как кражу интеллектуальной собственности. Его новые работодатели, зная об этом, вряд ли хорошо отреагируют, если он признается в использовании какого-либо старого кода.

Ньюби вполне может найти свой инструментарий бесполезным, даже если сможет украдкой внести его в сборку на новой работе. Его новые работодатели могут использовать другой набор коммерческих инструментов, языков и библиотек. Вероятно, ему придется изучать какую-либо новую группу методик и изобретать новый набор "колес" при каждой смене проекта.

Так программисты сталкиваются с повторным использованием кода (и другими хорошими практическими приемами, которые ему сопутствуют, например, модульностью и прозрачностью), которое извне систематически обусловливается комбинацией технических проблем, барьеров интеллектуальной собственности, политических и личных потребностей.

"Умножьте" Дж. Рэндома Ньюби на сто тысяч, "состарьте" его на пару десятков лет и дайте ему стать более циничным и привыкшим к данной системе. В результате получится описание большей части индустрии программного обеспечения, рецепт огромных потерь времени, средств и человеческого мастерства — даже

если не учитывать приемы маркетинговой тактики поставщиков, некомпетентный менеджмент, нереальные сроки завершения и все остальные факторы, которые усложняют качественное выполнение работы.

Профессиональная культура, происходящая из опыта Ньюби, ярко отражает этот опыт. Предприятия, разрабатывающие программное обеспечение, получат сильный NIH-комплекс (Not Invented Here — изобретено не здесь). Они будут упорно противостоять повторному использованию кода, навязывая своим программистам в целях соблюдения жестких проектных рамок неадекватные, но интенсивно продаваемые поставщиками компоненты, отвергая при этом повторное использование своего же протестированного кода. Они будут создавать множество узкоспециальных, дублируемых программ. Программисты, создающие эти программы, будут знать, что результатом будет свалка программ, но покорно смирятся с невозможностью исправить что-либо, кроме отдельных частей, созданных самостоятельно.

Вместо повторного использования кода будет создан догмат о том, что однажды купленный код нельзя выбрасывать, а напротив, следует исправлять его, даже если всем ясно, что было бы лучше избавиться от него и начать работу заново. Продукты такой культуры со временем постепенно будут становиться более раздутыми и полными ошибок, даже когда каждый вовлеченный индивидуум будет напряженно пытаться сделать свою работу хорошо.

16.2. Прозрачность — ключ к повторному использованию кода

История Дж. Рэндома Ньюби была испытана на множестве опытных программистов. Если читатель относится к их числу, то он, вероятно, отреагирует, как и большинство коллег: возгласом понимания. Если же читатель не является программистом, а управляет программистами, то автор искренне надеется, что история получилась назидательной.

Большинство из нас привыкли к исходным предположениям программной индустрии о том, что могут потребоваться значительные умственные усилия, прежде чем появится возможность выделить главные причины данной проблемы. Но в конечном итоге они не являются очень сложными.

В основе большинства неприятностей Дж. Рэндома Ньюби (и крупномасштабных проблем качества, которые они подразумевают) лежит прозрачность — или, вернее, ее недостаток. Невозможно устранить проблему в коде, который невозможно посмотреть изнутри. Фактически в любой программе с нетривиальным API-интерфейсом невозможно даже должным образом

использовать то, что нельзя увидеть изнутри. Документация неадекватна не только на практике, но и в принципе. Она не способна передать все нюансы, реализованные в коде.

В главе 6 отмечалось, как важна для хорошей программы центральная прозрачность. Компоненты, состоящие только из объектного кода, разрушают прозрачность программной системы. С другой стороны, неудачи повторного использования кода гораздо менее вероятно нанесут ущерб, когда повторно используемый код доступен для чтения и модификации. Хорошо комментированный исходный код сам по себе является документацией. Ошибки в исходном коде можно устранить. В исходный код можно включить средства измерения и

скомпилировать для отладки, что позволяет упростить проверку его поведения в непонятных случаях, а кроме того, всегда можно изменить поведение программы в случае необходимости.

Существует еще одна жизненно важная необходимость в исходном коде. Урок, который Unix-программисты изучали в течение десятилетий, заключается в том, что исходный код выдерживает испытание временем, а объектный — нет. Изменяются аппаратные платформы и служебные компоненты, такие как библиотеки поддержки, в операционных системах появляются новые API-интерфейсы и устаревают прежние. Меняется все, а непроницаемые двоичные исполняемые модули не способны адаптироваться к изменениям. Они хрупки, невозможно надежно обеспечить их дальнейшую переносимость, а кроме того, их необходимо поддерживать с помощью расширяющихся и подверженных ошибкам уровней кода эмуляции. Они замыкают пользователей в рамках предположений разработчиков. Исходный код необходим, ведь даже если нет ни намерений, ни необходимости изменять программу, в будущем для обеспечения работоспособности программы ее придется перекомпилировать в новых средах.

Важность прозрачности и проблема устаревания кода являются теми причинами, по которым следует требовать открытости повторно используемого кода для проверки и модификации [135]. Это лишь предварительный аргумент в пользу того, что сейчас называется "открытым исходным кодом". Понятие "открытый исходный код" имеет более важный смысл, чем просто требование прозрачности и видимости кода.

16.3. От повторного использования к открытому исходному коду

Компоненты ранней Unix, ее библиотеки и связанные утилиты распространялись в виде исходного кода. Эта открытость была жизненно важной частью культуры Unix. В главе 2 уже говорилось о том, как после разрушения этой традиции в 1984 году Unix утратила свою первоначальную движущую силу, и как десять лет спустя возникновение GNU-инструментария и Linux побудило сообщество "вернуться к ценностям" открытого исходного кода.

В настоящее время открытый исходный код снова является одним из наиболее мощных инструментов в коллекции любого Unix-программиста.

Понятие открытого исходного кода так же связано с повторным использованием кода, как романтическая любовь связана с размножением — описать первое в терминах второго возможно, но при этом существует риск упустить многое из того, что делает первое интересным. Понятие "открытый исходный код" не сводится к простой тактике поддержки повторного использования в разработке программ. Открытый исходный код — неожиданно возникающий феномен, общественное соглашение между разработчиками и пользователями, которые пытаются защитить несколько преимуществ, связанных с прозрачностью. Фактически существует несколько различных способов толкования данного феномена.

Ранее в настоящей книге историческое описание было подано в контексте причинных и культурных связей между Unix и открытым исходным кодом. Организация и тактика разработки открытого исходного кода рассматривается в главе 19. При обсуждении теоретических и практических вопросов повторного использования полезно рассматривать открытый исходный код отдельно, как непосредственную реакцию на проблемы, драматизированные в истории о Дж. Рэндома Ньюби.

Разработчики программного обеспечения хотят, чтобы код, который они используют, был

прозрачным. Более того, они не хотят терять свой инструментарий и опыт при переходе на новое место работы. Они устали быть жертвами, им надоело разочаровываться грубыми инструментами, границами интеллектуальной собственности, а также необходимостью многократно изобретать колесо.

Разработчики программного обеспечения похожи на всех остальных мастеров и изобретателей. Они . хотят быть художниками и не скрывают этого. У них есть внутренние стимулы и потребности художников, включая желание иметь аудиторию. Он не только хотят повторно использовать код, они также хотят, чтобы другие использовали их код повторно. В этом есть непреодолимое желание, которое выходит далеко за рамки краткосрочных экономических целей и упраздняет их, его невозможно удовлетворить, создавая программы с закрытым исходным кодом.

Открытый исходный код — основной идеологический удар по всем данным проблемам. Если корень большинства проблем Дж. Рэндома Ньюби с повторным использованием заключается в непрозрачности зарытого исходного кода, то первоначальные предположения, которые порождают закрытый исходный код, должны быть разрушены. Если корпоративная территориальность является проблемой, то ее необходимо атаковать или игнорировать до тех пор, пока корпорации не поймут, насколько губительными для них являются их территориальные рефлексы. Открытый исходный код — вот, что случится, когда повторное использование кода "получит флаг и армию".

Таким образом, с конца 90-х годов прошлого века более нет никакого смысла рекомендовать стратегию и тактику повторного использования кода без обсуждения открытого исходного кода, соответствующих практических приемов, вариантов лицензирования и норм сообщества открытого исходного кода. Даже если данные проблемы в других сообществах могли бы быть обособленными, в мире Unix они связаны неразрывно.

В оставшейся части данной главы рассматриваются различные проблемы, связанные с повторным использованием открытого исходного кода: оценка, документация и лицензирование. В главе 19 модель разработки открытого исходного кода обсуждается шире, а также рассматриваются соглашения, которых следует придерживаться при опубликовании кода для массового использования.

16.4. Оценка проектов с открытым исходным кодом

В Internet доступны для использования буквально терабайты исходных кодов для системных и прикладных Unix-программ, служебных библиотек, GUI-инструментариев и аппаратных драйверов. Большинство из них с помощью стандартных инструментов можно скомпилировать и запустить за считанные минуты: ./configure; make; make install; для выполнения инсталляционной части обычно требуются привилегии администратора.

Люди за пределами мира Unix (особенно нетехнические пользователи) склонны рассматривать программное обеспечение с открытым исходным кодом (или "свободное") как непременно худшее по сравнению с коммерческим, т.е. как низкокачественное, ненадежное и вызывающее больше проблем, чем способно решить. Они упускают из виду важный момент: в общем, программное обеспечение с открытым исходным кодом пишется людьми, которым оно не безразлично, которые сами его используют, и, опубликовывая его, рискуют своей личной репутацией. Они также склонны тратить меньше времени на совещания, давние изменения конструкции и бюрократические издержки. Следовательно, они сильнее мотивированы и лучше нацелены на выполнение качественной работы, чем оплачиваемые невольники, для которых главное — соблюсти невозможные сроки завершения проекта в

частных предприятиях.

Более того, сообщество пользователей открытого исходного кода не боится устранять ошибки, а стандарты этого сообщества высоки. Авторы, выдвигающие работу несоответствующую стандартам, сталкиваются с сильным общественным давлением, вынуждающим либо исправить код, либо удалить его, а при желании могут получить существенную квалифицированную помощь в исправлении. Как результат, зрелые пакеты программ с открытыми исходными кодами обычно имеют высокое качество и часто функционально превосходят любые коммерческие эквиваленты. Им может не доставать блеска, а в документации иногда много предположений, но жизненно важные части обычно работают совершенно четко.

Кроме эффекта коллегиальной оценки существует другая причина ожидать более высокого качества: в мире открытого исходного кода разработчики никогда не закрывают глаза на ошибки под давлением срока завершения проекта. Главное отличие между открытым и коммерческим кодами заключается в том, что версия 1.0 фактически означает готовое к использованию программное обеспечение. В действительности версия 0.90 или выше является довольно убедительным свидетельством того, что код готов к серийному использованию, но разработчики еще не вполне готовы утверждать это.

Читателю, который не является Unix-программистом, может быть трудно поверить в это. Еще один аргумент: в современных Unix-системах C-компилятор почти неизменно является программой с открытым исходным кодом. Коллекция GNU-компиляторов (GNU Compiler Collection — GCC) Фонда свободного программного обеспечения настолько мощная, хорошо документированная и надежная, что рынка для коммерческих Unix-компиляторов фактически не осталось, а для Unix-поставщиков стало нормой создавать версии GCC для своих платформ, не разрабатывая собственных компиляторов.

Способ оценки пакета открытого исходного кода заключается в прочтении его документации и беглом ознакомлении с некоторой частью самого кода. Если все увиденное производит впечатление компетентно написанного и внимательно документированного, то в нем можно быть уверенным. Если также имеются признаки того, что данный пакет какое-то время является популярным и значительно доработан с учетом откликов пользователей, то можно считать, что программа совершенно надежна (тем не менее, ее все-таки следует протестировать).

Благодарности многим людям за отправленные решения и исправления свидетельствуют о значительном контингенте пользователей, контактирующих с авторами, а также о том, что добросовестный куратор отвечает на замечания и принимает исправления.

Хорошим знаком является также наличие Web-страницы программы, списка часто задаваемых вопросов (Frequently Asked Questions — FAQ) и связанного списка почтовой рассылки или группы новостей Usenet. Все это подтверждает, что вокруг данной программы формируется настоящее сообщество заинтересованных пользователей. Недавние обновления Web-страниц и обширный список зеркал также являются надежными признаками проекта с жизнеспособным пользовательским сообществом. Бесполезные пакеты просто не получат такого длительного вклада, поскольку не способны его оправдать.

На разностороннюю пользовательскую базу указывают также версии программы для нескольких платформ.

Информацию о высококачественных программах с открытым исходным кодом можно найти на следующих Web-страницах.

• GIMP <http://www.gimp.org/>;

• GNOME <http://www.gnome.org>;

• KDE <http://www.kde.org>;

• Python <http://www.python.org>;

• Ядро Linux <http://www.kernel.org>;

• PostgreSQL <http://www.postgresql.org>;

• XFree86 <http://xfree86.org>;

• InfoZip <http://www.info-zip.org/pub/infozip/>.

Просмотр Linux-дистрибутивов — еще один хороший способ поиска качественных программ. Сборщики дистрибутивов Linux и других Unix-систем с открытым исходным кодом имеют большой опыт экспертной оценки лучших в своем роде проектов. Если читатель уже использует Unix-систему с открытым исходным кодом, имеет смысл проверить, включен ли оцениваемый пакет в состав дистрибутива.

16.5. Поиск открытого исходного кода

Ввиду того, что в Unix-мире доступно огромное количество открытого исходного кода, навыки поиска такого кода для повторного использования могут иметь неоценимое значение — гораздо большее, чем в случае других операционных систем. Существует множество форм такого кода: отдельные фрагменты и примеры кода, библиотеки кода, утилиты для повторного использования в сценариях. В Unix повторное использование в большинстве случаев является не фактическим копированием и вставкой кода в разрабатываемую программу — по сути, если разработчик действительно так поступает, то вполне вероятно, что существует не известный ему более элегантный способ повторного использования. Соответственно, одним из наиболее полезных навыков, которые следует культивировать при работе в Unix, является твердое понимание всех различных способов связывания кода так, чтобы можно было использовать правило композиции.

Для того чтобы найти код, который можно использовать повторно, начинать следует прямо с используемой системы. Unix-системы всегда характеризовались обширным инструментарием повторно используемых утилит и библиотек; более современные, такие как широко распространенная в настоящее время операционная система Linux, включают в себя тысячи программ, сценариев и библиотек, которые можно использовать повторно. Простой поиск в документации с помощью команды man -k с несколькими ключевыми словами часто дает полезные результаты.

Для того чтобы убедиться в удивительном богатстве ресурсов в Internet, следует посетить сайты проектов SourceForge, ibiblio и Freshmeat.net. В будущем могут появиться другие сайты такой же важности, однако указанные три в течение многих лет являются стабильно популярными, и, вероятно, останутся таковыми.

Проект SourceForge <http://www.SourceForge.net> — демонстрационный сайт для программ, специально предназначенный для поддержки совместной разработки и имеет связанные службы управления проектами. Данный проект представляет собой не просто архив, а бесплатную службу поддержки разработки, и в середине 2003 года, несомненно, являлся крупнейшим центром движения открытого исходного кода.

Архивы Linux на сайте ibiblio <http://www.ibiblio.org> до появления SourceForge были крупнейшими в мире. Архивы ibiblio являются пассивным хранилищем, т.е. просто местом для публикации пакетов. Однако они имеют лучший интерфейс к World Wide Web, чем большинство пассивных сайтов (программа, создающая вид и восприятие данного проекта в Web рассматривалась в качестве одного из учебных примеров при обсуждении Perl в главе 14). Данный сайт также является основным для проекта Linux Documentation Project, в рамках которого поддерживаются множество документов, являющихся замечательными ресурсами для Unix-пользователей и разработчиков.

Проект Freshmeat <http://www.freshmeat.net> является системой, специально предназначенной для публикации объявлений о выпуске нового программного обеспечения и новых версий старых программ. Он позволяет пользователям и третьим лицам писать обзоры версий программ.

Эти три универсальных сайта содержат код на многих языках, но большая часть опубликованного в них кода написана на C или C++. Существуют также сайты, специализирующиеся на каком-либо интерпретируемом языке, как было сказано в главе 14.

Архив CPAN — центральный репозиторий полезного бесплатного кода на Perl. Он доступен с домашней страницы проекта Perl <http://www.perl.com/perl>.

Сообщество Python-программистов (Python Software Activity) создает архив Python-программ и документации, доступной на домашней странице проекта Python, <http://www.python.org>.

Множество Java-аплетов и ссылок на другие сайты, содержащие бесплатные Java-программы, доступны на странице Java-аплетов <http://java.sun.com/applets/>.

Для Unix-разработчика также очень важен просмотр данных сайтов в целях определения того, что является доступным для использования. Это позволяет сэкономить время на кодирование.

Просмотр метаданных пакета является хорошей идеей, но не следует останавливаться на этом. Необходимо также испытать код. Это позволит полнее понять работу кода и впоследствии поможет эффективнее его использовать.

В более широком смысле чтение кода является инвестицией в будущее. Чтение кода учит новым методикам, новым способам разделения проблем, различным стилям и подходам. Как использование кода, так и его изучение, несомненно, обогащают разработчика ценным опытом. Даже если разработчик не применяет использованные в рассматриваемом коде методики, уточненное определение проблемы, которое он получает, рассматривая решения других, может серьезно помочь в создании собственного лучшего решения.

Читайте перед написанием кода; развивайте привычку чтения кода. Абсолютно новые проблемы встречаются редко, поэтому почти всегда существует возможность найти и использовать в качестве отправной точки код, который достаточно близок к тому, от чего можно оттолкнуться при решении проблемы. Даже если решаемая проблема действительно нова, вполне возможно, что она "генетически" связана с проблемой, которая ранее была решена другим разработчиком, поэтому необходимое решение, вероятно, также будет связано с каким-либо уже существующим решением.

16.6. Вопросы использования программ с открытым исходным кодом

При использовании или повторном использовании программного обеспечения с открытым исходным кодом определяются три главные проблемы: качество, документация и условия лицензирования. Выше отмечалось, что если разработчик не слишком придирчив к альтернативам, то, как правило, он находит одну или несколько альтернатив с приемлемым качеством.

Документация часто является более серьезным вопросом. Многие высококачественные пакеты с открытым исходным кодом, ввиду своей слабой документированности, гораздо менее полезны, чем хотелось бы. Традиции Unix строго определяют стиль документации, в соответствии с которым (несмотря на то, что она может технически охватить все функции пакета) предполагается, что читатель хорошо знаком с проблемной областью и читает очень внимательно. Для этого есть весомые причины, которые обсуждаются в главе 18, однако данный стиль может создавать некоторые препятствия. К счастью, извлечение из документации ценной информации относится к развиваемым навыкам.

Рекомендуется использовать Web-поиск по фразам, включающим в себя название программного пакета или тематические ключевые слова, а также строку "HOWTO" или "FAQ". Такие запросы часто позволяют найти более полезную для новичков информацию, чем man-страницы.

Самой серьезной проблемой при повторном использовании программного обеспечения с открытым исходным кодом (особенно в коммерческих продуктах любого вида) является понимание того, какие обязательства, накладывает лицензионное соглашение на разработчика, использующего данный код. В следующих двух разделах эта проблема рассматривается более подробно.

16.7. Вопросы лицензирования

Любое произведение, которое не является общедоступным, охраняется авторским правом, а возможно, даже не одним.

Закон об авторском праве не дает полного и четкого понятия о том, кого следует считать автором, особенно для программного обеспечения, которое создается стараниями многих людей. Именно по этой причине важны лицензии. Они могут санкционировать различные способы использования кода, которые в противном случае, согласно закону об авторском праве, были бы недопустимы. Лицензии, написанные соответствующим образом, способны защитить пользователей от судебного преследования со стороны правообладателей.

В мире коммерческого программного обеспечения лицензионные условия направлены на защиту авторских прав. Они характеризуют способ, гарантирующий пользователям только несколько прав, тогда как для владельца (правообладателя) резервируется как можно большая "легальная территория". Правообладатель в данном случае играет очень важную роль, а лицензионная логика настолько ограничивает использование, что точные технические подробности лицензионных условий обычно второстепенны.

Как будет отмечено ниже, правообладатель обычно использует авторское право для защиты лицензии, делающей код легкодоступным согласно условиям, которые правообладатель намерен сохранять неограниченное время. В противном случае резервируются только некоторые права, а пользователю предоставляется возможность выбора. В частности, правообладатель не может изменить условия использования той копии, которая уже имеется у пользователя. Таким образом, в программном обеспечении с открытым исходным кодом правообладатель почти несущественен, а условия лицензии очень важны.

Обычно правообладателем проекта является его нынешний лидер или финансирующая организация. Передача проекта новому лидеру часто обусловлена сменой правообладателя. Однако данная практика не является жестким правилом. Многие проекты с открытым исходным кодом имеют нескольких правообладателей, и пока не зафиксировано случаев возникновения правовых проблем. В некоторых проектах авторское право передается Фонду свободного программного обеспечения, который заинтересован в защите открытого исходного кода и имеет штат подготовленных юристов.

16.7.1. Что определяется как открытый исходный код

Лицензия может ограничивать или обусловливать любое из следующих прав: право на копирование и воспроизведение, право на использование, право модификации для персонального использования и право на воспроизведение модифицированных копий.

Определение открытого исходного кода (Open Source Definition) <http://www.opensource.org/osd.html> является результатом долгих размышлений о том, что делает программное обеспечение "открытым" (open source) или (в прежней терминологии) "свободным" (free). Оно широко принимается в сообществе открытого исходного кода как озвучивание общественной договоренности среди разработчиков открытого исходного кода. Его ограничения относительно лицензирования предполагают выполнение следующих требований:

• гарантия неограниченного права копирования;

• гарантия неограниченного права на воспроизведение в неизменной форме;

• гарантия неограниченного права на модификацию для персонального использования.

Данные принципы запрещают ограничение на воспроизведение модифицированных бинарных файлов, что отвечает потребностям дистрибьюторов программного обеспечения, которым необходимо без затруднений распространять работающий код. Это позволяет авторам требовать, чтобы модифицированные исходные коды распространялись как изначальный код плюс исправления к нему. Таким образом, определяются намерения автора и "контрольное отслеживание" любых изменений, внесенных другими.

OSD — легальное определение сертификационного знака "OSI-сертифицированного открытого исходного кода" (OSI Certified Open Source), а также лучшее из когда-либо созданных определений "свободного программного обеспечения". Все стандартные лицензии (MIT, BSD, Artistic, GPL/LGPL и MPL) соответствуют данному определению (хотя некоторые, такие как GPL, имеют другие ограничения, в условиях которых следует разобраться, прежде чем принимать их).

Необходимо отметить, что лицензии, позволяющие только некоммерческое использование, не квалифицируются как лицензии открытого исходного кода, даже если они базируются на GPL или любой другой стандартной лицензии. Такие лицензии дискриминируют определенные занятия, личности и группы, а такая практика недвусмысленно запрещается статьей 5 OSD.

Статья 5 данного документа была написана после нескольких лет болезненных экспериментов. Лицензии на некоммерческое использование столкнулись с проблемой отсутствия четко сформулированного правового теста для определения того, какой вид воспроизведения квалифицируется как "коммерческий". Безусловно, так квалифицируется

продажа программы как продукта. А если программа распространяется с нулевой номинальной ценой совместно с другой программой или данными, а цена "ложится грузом" на всю коллекцию? Что изменится, если данная программа существенна для работы всей коллекции?

Никто не знает. Сам факт, что лицензии на некоммерческое использование создали неопределенность в правовом поле редистрибьюторов, ставит под удар такие лицензии. Одной из целей создания документа OSD является гарантия того, чтобы люди в цепи распространения OSD-программ для ознакомления со своими правами не нуждались в консультациях правоведов в области интеллектуальной собственности. OSD запрещает сложные ограничения против лиц, групп и видов деятельности, с тем чтобы люди, имеющие дело с коллекциями программ, не столкнулись с проблемами несколько отличающихся (а возможно конфликтующих) ограничений на варианты использования продуктов.

Это беспокойство не является гипотетическим. Например, очень важной частью цепи распространения в сообществе открытого исходного кода являются CD-ROM-дистрибьюторы, которые собирают программы в полезные коллекции, начиная от простых сборников до загружаемых операционных систем. Должны быть запрещены ограничения, которые чрезмерно усложнили бы жизнь CD-ROM-дистрибьюторов или других дистрибьюторов, пытающихся распространять программное обеспечение с открытым исходным кодом на коммерческой основе.

С другой стороны, в OSD ничего не сказано о местном законодательстве. В некоторых странах действуют законы, препятствующие экспорту определенных технологий в другие указанные страны. OSD не может их опровергнуть, данный документ лишь указывает на то, что владельцы лицензий не могут добавлять собственных ограничений.

16.7.2. Стандартные лицензии на открытый исходный код

Ниже представлены стандартные условия лицензий в проектах с открытым исходным кодом, с которыми читателю, вероятно, придется столкнуться.

MIT <http://www.opensource.org/licenses/mit-license.html>

Лицензия MIT или Консорциума X (MIT X Consortium License— подобна лицензии BSD, но без рекламных требований).

BSD <http://www.openeource.org/licenses/bsd-license.html>

Авторское право членов правления Калифорнийского университета в Беркли (используется для BSD-кода).

Artistic License (Артистическая лицензия)
<http://www.opensource.org/licenses/artistic-license.html>

Те же условия, что и в Артистической лицензии Perl (Perl Artistic License).

GPL <http://www.gnu.org/copyleft.html>

Общедоступная лицензия GNU (GNU General Public License).

LGPL <http://www.gnu.org/copyleft.html>

"Library" (Библиотечная) или "Lesser" (Облегченная) GPL-лицензия.

MPL <http://www.opensource.org/licenses/MPL-1.1.html>

Общественная лицензия Mozilla (Mozilla Public License).

Данные лицензии более подробно (с точки зрения разработчика) обсуждаются в главе 19. В данной главе достаточно подчеркнуть, что единственное важное отличие между ними заключается в том, что лицензии могут быть переходящими и непереходящими. Лицензия является

переходящей (infectious) в случае, если она требует, чтобы любая производная работа лицензионной программы также попадала под условия данной лицензии.

Согласно данным лицензиям, единственным видом использования открытого исходного кода, действительно вызывающим беспокойство, является фактическое внедрение свободного кода в частный продукт (в отличие от, например, простого использования инструментов разработки с открытым исходным кодом для создания собственного продукта). Если разработчик не против включения необходимых лицензионных подтверждений и указателей на используемый исходный код в документацию по его продукту, то даже непосредственное внедрение кода должно быть безопасно при условии, что лицензия не является переходящей.

GPL является одновременно самой распространенной и самой спорной переходящей лицензией. В ней есть статья 2(b), которая требует, чтобы любая производная работа GPL-программы сама была лицензирована на условиях GPL, что вызывает разногласия. (К некоторым разногласиям привела статья 3(b), требующая, чтобы обладатели лицензий предоставляли по требованию исходный код на физических носителях, однако взрывной рост Internet сделал публикацию архивов исходного кода, как того требует пункт 3(a) лицензии, настолько дешевой, что никто более не заботится о требованиях опубликования.)

Никто не может точно сказать, что подразумевается под словами "содержит или является производной из..." в статье 2(b), или какие виды использования стоят за формулировкой "простое агрегирование" несколькими параграфами ниже. Спорные вопросы касаются компоновки библиотек и включения GPL-лицензированных файлов заголовков. Часть проблемы состоит в том, что законы США, касающиеся авторского права, не определяют понятия производной; эта задача оставлена судам для создания определений прецедентного права, а компьютерные программы являются той областью, в которой данный процесс начался совсем недавно.

С одной стороны, "простое агрегирование" определенно делает безопасным поставку GPL-программы на одном носителе с разработанным коммерческим кодом при условии, что они не связаны друг с другом или не вызывают друг друга. Они даже могут быть инструментами, оперирующими с одинаковыми файловыми форматами или дисковыми структурами; данная ситуация, согласно закону об авторском праве, не сделала бы одну программу производной от другой.

С другой стороны, внедрение GPL-кода в разрабатываемый коммерческий код или связывание объектного GPL-кода с коммерческим определенно делает последний производным продуктом и требует его лицензирования на условиях GPL.

Существует общее мнение о том, что одна программа может запускать другую как подчиненный процесс, и при этом ни одна из программ не становится производным продуктом другой.

Вызывающим споры случаем является динамическое связывание общих библиотек. Позиция

Фонда свободного программного обеспечения такова: если программа вызывает другую программу как общую библиотеку, то данная программа является производным продуктом библиотеки. Некоторые программисты считают данное утверждение уловкой. Существуют технические, правовые и политические аргументы в пользу обеих точек зрения; здесь они не рассматриваются. Поскольку Фонд свободного программного обеспечения является владельцем данной лицензии, было бы разумно поступать так, как будто позиция FSF правильна, до тех пор, пока суд не примет противоположное решение.

Некоторые считают, что формулировка статьи 2(b) умышленно направлена на "инфицирование" каждой части любой коммерческой программы, использующей даже небольшой фрагмент GPL-лицензированного кода. Они называют данную лицензию GPV (General Public Virus — общедоступный вирус). Другие считают, что формулировка "простое агрегирование" скрывает все недостатки "смеси" GPL и не-GPL-кода в одной компиляции или загрузочном модуле.

Эта неопределенность вызвала немалое возмущение в сообществе открытого исходного кода, и FSF пришлось разработать специальную, несколько менее жесткую версию "Library GPL" (которая затем была переименована в "Lesser GPL"), чтобы заверить людей в том, что они могут продолжать использовать динамические библиотеки, поставляемые с коллекцией GNU-компиляторов.

Разработчику приходится выбирать собственную интерпретацию статьи 2(b); большинство юристов не понимают связанных с ней технических вопросов, а прецедентного права не существует. Что касается практического опыта, FSF никогда (с момента своего основания в 1984 году и, по крайней мере, до середины 2003 года) не преследовал никого в судебном порядке за нарушение GPL, однако Фонд успешно во всех известных случаях усилил GPL угрозой судебного преследования. Известно также, что Netscape включает исходный и объектный код GPL-программы с коммерческим дистрибутивом браузера Netscape Navigator.

Переходность лицензий MPL и LGPL более ограничена, чем лицензии GPL. Они определенно позволяют связывание с частным кодом без превращения данного кода в производный продукт при условии, что весь трафик между GPL- и не-GPL-кодом проходит через библиотечный API или другой четко определенный интерфейс.

16.7.3. Когда потребуется адвокат

Данный раздел предназначен для коммерческих разработчиков, рассматривающих внедрение в закрытые продукты программ, подпадающих под условия одной из описанных стандартных лицензий.

Разобравшись во всем этом правовом словоблудии, очевидно, придется признать неутешительный факт — разработчики не юристы, и если есть какие-либо сомнения относительно легальности планируемого использования программ с открытым исходным кодом, то следует немедленно проконсультироваться у адвоката.

Правда, адвокаты и судьи запутываются в формулировках лицензий больше, чем разработчики. Законодательство о правах на программное обеспечение туманное, а прецедентного права по лицензиям на открытый исходный код (к середине 2003 года) не существовало; никто никогда не преследовался за нарушение данных лицензий.

Все это означает, что адвокат вряд ли значительно лучше понимает лицензии, чем внимательный читатель без юридической подготовки. Однако адвокаты остерегаются

непонятных им вещей. Поэтому если попросить совета у одного из них, то он, вероятнее всего, порекомендует держаться в стороне от программ с открытым исходным кодом, несмотря на тот факт, что он, возможно, не понимает технических аспектов или намерений автора так, как их понимает разработчик.

Наконец, разработчики, публикующие свою работу под лицензиями открытого исходного кода, обычно не являются представителями крупных корпораций, обслуживаемых множеством адвокатов. Это отдельные лица или группы добровольцев, которые главным образом хотят дарить свои программы. Крупные компании, выпускающие продукты как под данными лицензиями, так и за деньги, широко используют открытый исходный код и не хотят противостоять сообществу разработчиков, которое его производит, создавая правовые неприятности. Следовательно, вероятность попасть под суд за невинное техническое нарушение весьма невысока.

Однако это не означает, что данные лицензии можно игнорировать. Это было бы неуважением к творчеству и труду людей, создающих программы. А кроме того, неприятно стать первой мишенью судебного преследования со стороны разъяренного автора независимо от того, как будет протекать процесс.

Часть IV

Сообщество

17

Переносимость: переносимость программ и соблюдение стандартов

Осознание того, что операционные системы целевых машин были настолько же большим препятствием для переносимости, насколько их аппаратная архитектура, привело нас к радикальному предложению: избежать этой проблемы путем переноса самой операционной системы.

Переносимость C-программ и операционной системы Unix (Portability of C Programs and the UNIX System, 1978) –Стив Джонсон, Деннис Ритчи

Unix была первой действующей операционной системой, переносимой между различными семействами процессоров (Version 6 Unix, 1976 - 1977). В настоящее время Unix регулярно переносится на все новые машины, мощность которых достаточна для поддержки блока управления памятью. Unix-приложения просто переносятся между Unix-системами, работающими на различном аппаратном обеспечении; фактически, неудачные попытки переноса не известны.

Переносимость всегда была одним из принципиальных преимуществ Unix.
Unix-программисты склонны писать программы, основываясь на предположении, что аппаратное обеспечение изменчиво, а стабилен только Unix API, и делая как можно меньше предположений о специфических характеристиках машин, таких как длина машинного слова, порядок следования байтов или архитектура памяти. Фактически код, так или иначе

зависящий от аппаратного обеспечения, который выходит за пределы абстрактной машинной модели языка C, считается в Unix-кругах плохим и действительно допускается только в очень специфических случаях, таких, например, как ядра операционных систем.

Unix-программисты знают по опыту, что очень легко ошибиться, предрекая короткую жизнь программному проекту[136]. Поэтому они склонны избегать создания программного обеспечения, зависимого от специфических и недолговечных технологий, и в большой степени полагаются на открытые стандарты. Привычка писать программы с учетом переносимости уже стала одной из традиций Unix, так что теперь она относится даже к мелким проектам, которые задумывались как однократно используемый код. Она оказала влияние на всю конструкцию инструментария Unix-разработчика, а также на языки программирования, такие как Perl, Python и Tcl, которые создавались в Unix-среде.

Непосредственный выигрыш от переносимости заключается в том, что для Unix-программ является нормой пережить свою исходную аппаратную платформу, поэтому не требуется каждые несколько лет заново изобретать инструменты и приложения. Сегодня приложения, первоначально написанные для Version 7 Unix (1979), регулярно используются не только на Unix-системах, "генетически" происходящих от V7, но и на таких вариантах системы, как Linux, в которой API-интерфейс операционной системы был написан на основании Unix-спецификации и не содержит кода, заимствованного у Bell Labs.

Косвенная выгода менее очевидна, но, возможно, более важна. Дисциплина переносимости склонна упрощать архитектуры, интерфейсы и реализации. Это увеличивает вероятность успешного проекта и сокращает затраты на обслуживание в течение жизненного цикла программ.

В данной главе рассматривается диапазон и история Unix-стандартов. Ниже обсуждается, какие стандарты до сих пор актуальны, и описываются области большего или меньшего расхождения в Unix API. Кроме того, здесь рассматриваются инструменты и практические приемы, которые используются Unix-разработчиками для сохранения переносимости кода, а также формулируются некоторые важные принципы хорошей практики.

17.1. Эволюция C

Главным фактом практики Unix-программирования всегда была стабильность языка C и небольшого количества служебных интерфейсов, которые всегда ему сопутствовали (особенно стандартная I/O-библиотека и подобные ей). Тот факт, что язык, созданный в 1973 году, в течение тридцати лет интенсивного использования потребовал небольших изменений, является действительно примечательным, в этом отношении язык C не имеет аналогов в компьютерной науке.

В главе 4 приводились аргументы в пользу того, что C достиг успеха, благодаря тому, что он выполняет роль тонкого связующего уровня над аппаратным обеспечением компьютера, приближающимся к "стандартной архитектуре" [4]. Однако для того чтобы понять другие факторы, необходимо кратко рассмотреть историю данного языка.

17.1.1. Ранняя история C

Язык C родился в 1971 году как язык системного программирования для PDP-11-варианта Unix и основывался на раннем интерпретаторе языка B, разработанного Кеном Томпсоном. Язык B, в свою очередь, был смоделирован с базового языка общего программирования (Basic Common Programming Language — BCPL), разработанного в Кембриджском университете в 1966–1967 годах[137].

Первоначальный C-компилятор Денниса Ритчи (который часто называли "DMR" по инициалам его создателя) обслуживал сообщество, быстро разрастающееся вокруг операционной системы Unix версий 5, 6 и 7. От шестой версии C языка произошла версия C компании Whitesmiths, повторная реализация, которая стала первым коммерческим C-компилятором и ядром IDRIS, первого клона Unix. Однако самые современные реализации C смоделированы на основе "переносимого C-компилятора" (Portable C Compiler — PCC) Стивена Джонсона (Steven C. Johnson), который дебютировал в седьмой версии и полностью заменил компилятор DMR как в ветви System V, так и в BSD-версиях четвертого поколения.

В 1976 году в шестой версии C были представлены объявления typedef, union и unsigned int. Также подверглись изменениям утвержденный синтаксис для инициализации переменных и некоторые составные операторы.

Оригинальным описанием языка C была книга Брайана Кернигана и Денниса Ритчи

"The C Programming Language", которую также называли "Белой книгой" [42]. Она была опубликована в 1978 году, в том же году появился C-компилятор Whitemiths.

Белая книга описывала усовершенствованную шестую версию языка C с одним значительным исключением, которое касалось обработки общедоступных переменных. Первоначально Ритчи намеревался смоделировать правила C с COMMON- объявлений языка FORTRAN, основываясь на теоретическом соображении, что любая машина, способная поддерживать FORTRAN, будет готова к использованию C. В модели с общим блоком общедоступная переменная может быть объявлена несколько раз; идентичные объявления объединяются компоновщиком. Однако два ранних варианта C (для мэйнфреймов Honeywell и IBM 360) работали на машинах с очень ограниченной общей памятью или примитивным компоновщиком, или имели оба недостатка. Таким образом, компилятор Version 6 C был перенесен в боле строгую модель определения (которая требовала максимум одно определение любой заданной общедоступной переменной и ключевого слова extern, задающего на нее ссылки), описанную в [42].

Это решение было отменено в C-компиляторе, который поставлялся с Version 7, после выяснения того, что большое количество существующего исходного кода зависело от более свободных правил. Под давлением обратной совместимости закончилась неудачей еще одна попытка перестроиться (в версии System V Release 1 1983 года), перед тем как в 1988 году в проекте стандарта ANSI (ANSI Draft Standard) окончательно установились правила определения. Общий блок внешних переменных до сих пор является общепризнанным отклонением от стандарта.

В седьмой версии C была введена конструкция enum, а значения struct и union интерпретировались как объекты первого класса, которые можно назначать, передавать в качестве аргументов и возвращать из функций (вместо передачи по адресу).

Другим главным изменением в V7 было то, что Unix-объявления структур данных теперь документировались в файлах заголовков и подключались. В предыдущих версиях Unix структуры данных (например, для каталогов) фактически печатались в руководстве, из которого программисты копировали их в свой код. Излишне говорить, что это было значительной проблемой для переносимости. Стив Джонсон.

Версия System III C компилятора PCC (которая также поставлялась с BSD 4.1c) изменила

обработку объявлений struct так, чтобы члены с одинаковыми именами в различных структурах не конфликтовали. Также были внесены объявления void и unsigned char. Область действия extern-объявлений локальных для функций была ограничена данной функцией и более не включала всего следующего за ней кода.

В проекте стандарта ANSI C (ANSI C Draft Proposed Standard) были добавлены определения const (для памяти только для чтения) и volatile (для таких ячеек, как отображаемые на память регистры ввода/вывода, которые можно модифицировать асинхронно из потока управления программы). В целях применения к любому типу был обобщен модификатор типа unsigned, а также добавлен идентификатор signed. Был добавлен синтаксис для auto-массива, инициализаторов структур, а также типов union. Наиболее важно то, что были добавлены прототипы функций.

Самыми важными изменениями в раннем C был переход к определению и введение прототипов функций в проекте стандарта ANSI C. Язык по существу остается стабильным с тех пор, как в 1985 - 1986 годах копии рабочих документов комитета X3J11 по проекту стандарта довели намерения комитета до сведения разработчиков компиляторов.

Более подробно история раннего C описана создателем языка в статье [70].

17.1.2. Стандарты C

Разработка стандартов C была консервативным процессом, в котором серьезное внимание уделялось "сохранению духа" оригинального C, а акцент был сделан скорее на утверждении экспериментов в существующих компиляторах, чем на создании новых функций. Документ-хартия C9X (C9X charter)[138] является превосходным выражением данной миссии.

Работа над первым официальным стандартом C началась в 1983 году при содействии комитета X3J11 ANSI. Главные функциональные дополнения к языку были утверждены к концу 1986 года, и с этого момента программисты стали различать "K&R C" и "ANSI C".

Многие не осознают, насколько

необычным, особенно оригинальная работа ANSI C, был проект стандартизации языка в своем упорном требовании стандартизировать только проверенные функции. Большинство комитетов по стандартизации языков тратят большую часть времени на создание новых функций, зачастую уделяя мало внимания их возможной реализации. Фактически несколько функций ANSI C, которые

были созданы с нуля, например, печально известные "триграфы" (trigraphs), были самыми непопулярными и наименее успешными функциями C89.Генри Спенсер.

Пустые указатели были созданы как часть работы по стандартизации и стали успешными. Но точку зрения Генри принимают до сих пор. Стив Джонсон.

Несмотря на то, что основа ANSI C была согласована достаточно рано, споры о содержимом стандартных библиотек продолжались в течение нескольких лет. Формальный стандарт не был опубликован до конца 1989 года, сразу после этого в большинстве компиляторов были реализованы рекомендации 1985 года. Данный стандарт первоначально назывался ANSI X3.159, но был переименован в ISO/IEC 9899:1990, когда Международная организация по стандартизации (International Standards Organization — ISO) приняла на себя спонсорские обязательства. Описываемый в стандарте вариант языка обычно называется C89 или C90.

Первая книга по C и практической реализации переносимости,

"Portable C and Unix Systems Programming" [44], была опубликована в 1987 году. (Я написал ее под корпоративным псевдонимом, навязанным мне моими тогдашними работодателями.) Второе издание книги Кернигана и Ритчи [42] вышло в 1988 году.

Очень незначительная модификация C89, названная Amendment 1 (поправка 1), AM1, или C93, появилась в 1993 году. В ней было добавлено больше возможностей по поддержке широких символов и Unicode. Данный стандарт получил название ISO/IEC 9899-1:1994.

Пересмотр стандарта C89 начался в 1993 году. В 1999 году организация ISO приняла стандарт ISO/IEC 9899 (обычно называемый C99). Он включал в себя Amendment 1 и большое количество мелких функций. Возможно, наиболее значительной из них для большинства программистов является возможность объявлять переменные в любом месте блока, а не только в начале (аналогичная языку C++). Кроме того, в стандарте были добавлены макросы с переменным числом аргументов.

Рабочая группа C9X имеет Web-страницу <http://anubis.dkuug.dk/JTC1/SC22/WG14/www/projects>, но к середине 2003 года планов на третий проект стандарта не было. Члены рабочей группы разрабатывают дополнения к C для встроенных систем.

Стандартизации C значительно способствовал тот факт, что работающие и почти полностью совместимые реализации языка функционировали на множестве различных систем до того, как началась работа по стандартизации. Это сильно ограничило споры о том, какие функции должны присутствовать в стандарте.

17.2. Стандарты Unix

Тот факт, что в 1973 году Unix была переписана на C, беспрецедентно упростил переносимость и модификацию операционной системы. В результате исходная Unix вскоре разделилась на семейство операционных систем. Unix-стандарты первоначально разрабатывались в целях согласования API-интерфейсов различных ветвей фамильного дерева.

Unix-стандарты, развившиеся после 1985 года, были весьма успешными в этом отношении — настолько успешными, что легли в основу высоко ценимой документации по API современных реализаций Unix. Фактически реальные Unix-системы так близко придерживаются опубликованных стандартов, что разработчики могут (а часто так и делают) узнать больше, изучая такие документы, как спецификация POSIX, чем официальные справочные руководства для используемого варианта Unix.

Действительно, в наиболее новых Unix-системах с открытым исходным кодом (таких как Linux) широко распространена практика проектирования функций операционной системы с использованием опубликованных стандартов как спецификации. Этот момент рассматривается в ходе изучения стандартов RFC далее в настоящей главе.

17.2.1. Стандарты и Unix-войны

Первоначальным мотивом для разработки Unix-стандартов было расхождение линий разработки AT&T и Университета Беркли, которое рассматривалось в главе 7.

Unix-системы 4.x BSD происходили от Version 7, вышедшей в 1979 году. После выхода 4.1 BSD в 1980 году BSD-линия быстро приобрела репутацию революционной Unix-системы. В число важных дополнений входили визуальный редактор

vi, средства контроля заданий для управления с одной консоли многочисленными высокоприоритетными и фоновыми задачами, а также усовершенствование сигналов (см. главу 7). Несравненно более важным дополнением была поддержка TCP/IP-сетей, но хотя Университет Беркли получил контракт на их реализацию в 1980 году, протокол TCP/IP в течение трех лет не включался в состав внешних версий.

Однако другая версия, System III, появившаяся в 1981 году, стала основой дальнейшей разработки в корпорации AT&T. В System III интерфейс терминалов версии 7 был переработан в более четкую и изящную форму, которая была абсолютно несовместимой с усовершенствованиями Беркли. В ней сохранялась (необратимая) семантика сигналов (см. главу 7). В состав январской версии System V Release 1 в 1983 году вошли некоторые BSD-утилиты (такие как

vi(1)).

Первая попытка заполнить разрыв была предпринята в феврале 1983 года влиятельной группой Unix-пользователей, UniForum. Вышедший в 1983 году проект стандарта Uniforum (Uniforum 1983 Draft Standard — UDS 83) описывал "основную Unix систему", состоящую из подмножества ядра System III и библиотек плюс примитив блокировки файлов. AT&T объявила о поддержке UDS 83, однако данный стандарт представлял собой неадекватный набор разрабатываемых практических приемов 4.1BSD. Проблема обострилась с выходом в июле 1983 года 4.2BSD, в которой было добавлено множество новых функций (включая TCP/IP-сети) и введена некоторая неочевидная несовместимость с исходной версией 7.

Лишение прав компаний Bell в 1984 году и начало Unix-войн (см. главу 2) значительно усложнило проблемы. Sun Microsystems лидировала в индустрии рабочих станций в BSD-направлении; AT&T пыталась внедриться в компьютерный бизнес и использовать контроль над Unix как стратегическое оружие, даже продолжая лицензировать операционную систему таким конкурентам, как Sun. Все поставщики принимали бизнес-решения дифференцировать свои версии Unix для получения конкурентного преимущества.

В ходе Unix-войн техническая стандартизация стала точкой зрения, которую отстаивали сотрудничающие технические специалисты, а большинство менеджеров по продуктам принимали ее неохотно или активно сопротивлялись. Одним крупным и важным исключением была корпорация AT&T, которая, анонсируя в январе 1984 года System V Release 2 (SVr2), объявила о намерении сотрудничать с группами пользователей по формированию стандартов. Второй пересмотр проекта стандарта UniForum в 1984 году проложил путь и одновременно повлиял на API SVr2. Позднее Unix-стандарты также стремились следовать System V, кроме тех областей, где BSD-средства имели очевидное функциональное превосходство (поэтому, например, современные стандарты Unix описывают средства управления терминалами System V, вместо BSD-интерфейса к тем же средствам).

В 1985 году AT&T опубликовала документ

System V Interface Definition (SVID), в котором содержалось более формальное описание SVr2 API, включающего стандарт UDS 84. Более поздние модификации SVID2 и SVID3 наметили курс на создание интерфейсов System V версий 3 и 4. Документ SVID стал основой POSIX-стандартов, которые в конечном итоге склонили большую часть спора Беркли/AT&T по поводу системных и библиотечных вызовов C в сторону подхода

AT&T.

Однако это не стало очевидным в течение еще нескольких лет. Тем временем Unix-войны были в разгаре. Например, в 1985 году вышли два конкурирующих API-стандарта для файловой системы с совместным доступом по сети: Network File System (NFS) корпорации Sun и AT&T-система Remote File System (RFS). NFS-система выиграла, поскольку Sun была готова разделить с другими не только спецификации, но и открытый исходный код.

Это был настоящий успех, потому что по своему логическому построению система RFS была все-таки лучшей моделью. Она поддерживала лучшую семантику блокировки файлов и лучшую идентификацию пользователей в различных системах и, в отличие от NFS, как правило, была направлена на точное получение более мелких деталей семантики файловой системы Unix. Однако данный урок был проигнорирован, даже когда в 1987 году его повторила система X Window с открытым исходным кодом, победив частную систему Sun Networked Window System (NeWS).

После 1985 года главные вопросы по Unix-стандартизации решались Институтом инженеров электротехники и электроники (Institute of Electrical and Electronic Engineers — IEEE). Комитет 1003 IEEE разработал серию стандартов, которые широко известны как POSIX[139]. Данные стандарты выходили за рамки простых системных вызовов и библиотечных средств C. Они определяли подробную семантику оболочки и минимальный набор команд, а также детальные привязки для различных языков программирования, отличных от C. Первая версия появилась в 1990 году, а вторая редакция была опубликована в 1996 году. Международная организация по стандартизации (International Standards Organization — ISO) приняла данные стандарты под названием ISO/IEC 9945.

Ниже перечислены некоторые из ключевых POSIX-стандартов.

1003.1 (опубликован в 1990 году)

Библиотечные процедуры. Описание API системных вызовов C, очень похожее на Version 7, кроме сигналов и интерфейса управления терминалами.

1003.2 (опубликован в 1992 году)

Стандартная оболочка и утилиты. Семантика оболочки строго повторяет семантику Bourne shell в System V.

1003.4 (опубликован в 1993 году)

Unix-система реального времени. Бинарные семафоры, блокировка памяти процесса, файлы с распределением памяти, приоритетное расписание, сигналы реального времени, циклы и таймеры, передача IPC-сообщений, синхронизированный I/O, асинхронный I/O, файлы реального времени.

Во втором издании 1996 года стандарт 1003.4 был разделен на 1003.1b (система реального времени) и 1003.1c (параллельные процессы).

Несмотря на то, что некоторые ключевые области, такие как семантика обработки сигналов и упущение BSD-сокетов, были недоопределены, первоначальные POSIX-стандарты стали основой всей последующей работы по стандартизации Unix. Они до сих пор цитируются как авторитетные, хотя и косвенно, посредством таких справочников, как

"POSIX Programmer's Guide" [47]. Де-факто стандарт Unix API до сих пор остается "POSIX плюс сокеты", более поздние стандарты, главным образом, добавляют функции и более точно определяют согласование в необычных крайних случаях.

Следующим игроком на этом поле была группа X/Open (позднее переименованная в Open Group), консорциум поставщиков Unix, сформированный в 1984 году. Стандарты данной организации X/Open Portability Guides (XPGs) первоначально развивались параллельно с проектами POSIX, а затем после 1990 года стандарты XPGs были включены в POSIX и расширили его. В отличие от POSIX, который представлял собой попытку сбора надежного подмножества из всех Unix-систем, стандарты XPGs больше склонялись к общей практике в наиболее развитом участке исследований; даже XPG1 в 1985 году, охватывающий системы SVr2 и 4.2BSD, включал в себя сокеты.

В 1987 году в стандарт XPG2 был добавлен API-интерфейс поддержки терминалов, которым, по существу, была библиотека

curses(3) системы System V. XPG3 в 1990 году влился в X11 API. XPG4 в 1992 году полностью утвердил полное соответствие стандарту 1989 года ANSI C. В стандартах XPG2, 3 и 4 интенсивно рассматривалась поддержка интернационализации и описывался сложный API для обработки наборов кодов и каталогов сообщений.

В материалах по стандартам Unix могут встретиться ссылки на стандарты "Spec 1170" (1993 года), "Unix 95" (1995 года) и "Unix 98" (1998 года). Это были сертификационные знаки стандартов X/Open, и в настоящее время они представляют только исторический интерес. Однако работа над стандартом XPG4 превратилась в спецификацию Spec 1170, которая стала первой версией Единой спецификации Unix (Single Unix Specification, SUS).

В 1993 году 75 поставщиков систем и программного обеспечения, включая все главные Unix-компании, поставили окончательную точку в Unix-войнах, когда объявили о финансовой поддержке X/Open в разработке общего определения Unix. Как часть данного соглашения X/Open приобрела права на торговую марку Unix. Объединенный стандарт стал называться Single Unix Standard (Единый стандарт Unix) версии 1. Версия 2 вышла в 1997 году. В 1999 году деятельность по стандарту POSIX перешла к X/Open.

В 2001 году группа X/Open (сейчас The Open Group) выпустил Единый стандарт Unix версии 3 (Single Unix Standard version 3) <http://www.Unix.org/version3/>. Все "потоки" стандартизации Unix API были, наконец, объединены в одно целое. Различные вариации Unix воссоединились на основе общего API. И это было встречено с большим энтузиазмом, по крайней мере, среди давних поклонников Unix, которые помнили о бурях 80-х годов.

17.2.2. Влияние новых Unix-систем

К сожалению, была одна неудобная деталь — поставщики Unix старой школы, финансирующие данный проект, находились под сильным влиянием со стороны Unix-систем новой школы с открытым исходным кодом, а в некоторых случаях они даже отказывались (в пользу Linux) от коммерческих Unix-систем, ради которых они потратили так много усилий для достижения надежного соответствия.

Тестирование соответствия, необходимое для проверки соответствия Единой спецификации Unix, — дорогая проблема. Оно должно осуществляться для каждого дистрибутива, но не учитывает многообразия дистрибьюторов операционных систем с открытым исходным кодом. В любом случае, Linux изменяется столь быстро, что любая версия дистрибутива, вероятно, устарела бы к моменту окончания сертификации[140].

Стандарты, подобные Single Unix Specification, не утратили полностью своей значимости. Они все еще являются ценными руководящими принципами для разработчиков Unix. Однако

далее стоит рассмотреть то, как "The Open Group" и другие организации, связанные с процессом стандартизации Unix старой школы, будут приспосабливаться к быстрому темпу выхода версий с открытым исходным кодом (а также к малобюджетному или безбюджетному функционированию групп разработчиков программного обеспечения с открытым исходным кодом).

17.2.3. Стандарты Unix в мире открытого исходного кода

В середине 1990-х годов сообщество открытого исходного кода начало собственную работу по стандартизации. Эти усилия основывались на совместимости на уровне кода, закрепленной стандартом POSIX и его потомками. В частности Linux, была написана с нуля способом, который зависел от доступности таких стандартов Unix API, как POSIX[141].

В 1998 году компания Oracle перенесла на Linux свой лидирующий на рынке баз данных продукт, этот шаг справедливо рассматривается как главный прорыв в движении принятия Linux. Ответственный инженер проекта, в подтверждение того, что API-стандарты выполнили свою задачу, на вопрос репортера о том, какие технические трудности пришлось преодолеть Oracle ответил: "Мы вводили make".

Таким образом, проблемой для Unix-систем новой школы была не API-совместимость на уровне исходного кода. Все принимают как должное возможность переносить исходный код между различными Linux, BSD и коммерческими дистрибутивами Unix без необходимости тратить серьезные усилия на обеспечение переносимости кода. Новой проблемой была не совместимость исходного кода, а бинарная совместимость. Почва под Unix неожиданно зашаталась вследствие массового спроса на аппаратное обеспечение PC.

В прежние времена каждая Unix-система работала на той аппаратной платформе, которая фактически была ее собственной. Было достаточно много различных наборов инструкций процессоров и аппаратных архитектур, на которые необходимо было переносить приложения на уровне исходного кода, чтобы вообще их перенести. С другой стороны, существовало сравнительно небольшое число основных версий Unix, каждая с относительно длительным сроком службы. Поставщики приложений, такие как Oracle, могли позволить себе затраты на компиляцию и распространение отдельных бинарных дистрибутивов для каждой из трех или четырех комбинаций аппаратно-программного обеспечения, поскольку они могли распределить затраты на перенос исходного кода среди большого числа клиентов и в течение достаточно долгого жизненного цикла продуктов.

Но затем поставщиков мини-компьютеров и рабочих станций захлестнул волна недорогих персональных компьютеров на основе процессоров 386-й серии, и Unix-системы с открытым исходным кодом изменили правила игры. Поставщики обнаружили, что они более не имеют стабильной платформы для поставки своих бинарных дистрибутивов.

Внешней проблемой, на первый взгляд, было большое количество дистрибутивов Unix, но по мере консолидации рынка Linux-дистрибутивов, становилось ясно, что реальной проблемой была скорость изменения. API-интерфейсы были стабильны, но ожидаемое расположение файлов системного администрирования, утилит, и таких элементов, как префиксы путей к пользовательским почтовым ящикам и файлам системных журналов, продолжало изменяться.

Первым проектом по стандартизации, который развился внутри нового сообщества Linux и BSD (начиная с 1993 года) был Стандарт иерархии файловой системы (Filesystem Hierarchy Standard — FHS). Он был включен в состав Базы стандартов Linux (Linux Standards Base —

LSB), которая также стандартизировала ожидаемый набор служебных библиотек и вспомогательных приложений. Оба стандарта стали результатом деятельности Группы свободных стандартов (Free Standards Group) <http://www.freestandards.org/>, которая к 2001 году стала играть роль аналогичную позиции консорциума X/Open среди Unix-поставщиков старой школы.

17.3. IETF и процесс RFC-стандартизации

Когда Unix-сообщество соединилось с культурой Internet-инженеров, в него также проникло мышление, сформированное процессом RFC-стандартизации IETF (Internet Engineering Task Force — Инженерная группа по решению конкретной задачи в Internet). Согласно традиции IETF, стандарты должны возникать из опыта с работающим прототипом, но как только они

становятся стандартами, код, который им не соответствует, считается нерабочим и безжалостно уничтожается.

К сожалению, обычно стандарты разрабатываются иначе. История вычислительной техники полна примеров того, как технические стандарты создавались в ходе процесса, объединявшего в себе худшие черты "грубой" философии и темной закулисной политики, что приводило к созданию спецификаций, которые не способны были повторить какие-либо существующие реализации. Еще хуже то, что многие из них были либо настолько требовательными, что практически не могли быть реализованными, либо настолько недоработанными, что создавали больше проблем, чем решали. Затем они пропагандировались среди поставщиков, которые при любом удобном случае игнорировали их, если они создавали неудобства.

Одним из самых печально известных примеров абсурдной стандартизации были сетевые протоколы Открытого взаимодействия систем (Open Systems Interconnect, OSI), которые недолго конкурировали с TCP/IP в 1980-х годах — семиуровневая модель на расстоянии выглядела изящной, но оказалась чрезмерно сложной и нереализуемой на практике[142]. Другим широко известным негативным примером является стандарт ANSI X3.64 для средств видеотерминалов, испорченный едва различимой несовместимостью между юридически согласованными реализациями. Даже после того, как символьные терминалы почти были вытеснены растровыми дисплеями, несовместимость продолжала создавать проблемы (в частности, именно поэтому функциональные и специальные клавиши в программе

xterm(1) иногда нарушают ее работу). Стандарт RS232 для последовательных соединений был настолько недоопределенным, что иногда казалось, будто не существует двух одинаковых последовательных кабелей. Для описания всех подобных отрицательных примеров стандартов, возможно, потребовалась бы книга такого же объема.

Существует известная фраза, которая обобщает философию IETF: "Мы отвергаем королей, президентов и голосование. Мы верим в суровое единодушие и работающий код"[143]. Данное требование существования работающей реализации предохранило ее от грубейших ошибок. В действительности критерий еще строже.

Прежде чем проектная спецификация будет принята в качестве Internet-стандарта, она должна быть реализована и протестирована несколькими независимыми сторонами на способность корректно работать и взаимодействовать, а также использоваться в еще более требовательных средах. Процесс Internet-стандартизации (The Internet Standards Process) — третья редакция (RFC 2026).

Все IETF-стандарты проходят стадию документов RFC (Requests for Comment — запросы на комментарии). Процесс подачи RFC умышленно неформален. RFC-документы могут предлагать стандарты, результаты исследований, формулировать философские обоснования для последующих RFC-документов или даже представлять собой шутки. Появление ежегодного первоапрельского RFC является ближайшим эквивалентом соблюдения "религиозного праздника" среди Internet-хакеров, что привело к возникновению таких перлов, как

"A Standard for the Transmission of IP Datagrams on Avian Carriers" (RFC 1149, стандарт для передачи IP-дейтаграмм с голубиной почтой) [144],

"The Hyper Text Coffee Pot Control Protocol" (RFC 2324, гипертекстовый протокол управления кофеваркой) [145] и

"The Security Flag in the IPv4 Header" (RFC3514, защитный флаг в заголовке IPv4) [146].

Однако шуточные RFC-документы —

единственный вид документов, которые немедленно становятся RFC. Серьезные предложения в действительности начинают свой путь как "Internet-Drafts" (Internet-черновики), которые распространяются для публичного комментирования через IETF-каталоги на нескольких широко известных узлах. Отдельные Internet-черновики не имеют формального статуса и могут быть изменены или удалены их создателями в любое время. Черновики, которые не были отозваны, но и не получили статус RFC, удаляются по истечении шести месяцев.

Internet-черновики не являются спецификациями, и разработчики, а также поставщики программного обеспечения особенно отказываются соглашаться с ними. Internet-черновики являются центральными точками обсуждения, обычно в рабочей группе, члены которой сообщаются посредством почтовой рассылки. Когда руководство рабочей группы сочтет целесообразным, Internet-черновик представляется на рассмотрение RFC-редактору для присвоения номера RFC.

Internet-черновик, опубликованный с RFC-номером, является спецификацией, с которой могут согласиться разработчики. Ожидается, что авторы RFC-документа и сообщество в целом начнут корректировать данную спецификацию на основании практического опыта.

Некоторые RFC-документы не продвигаются дальше этой стадии. Спецификация, которая не смогла привлечь внимание и не прошла реальных испытаний, может быть быстро позабыта, и в конечном счете маркируется RFC-редактором как "Not recommended" (не рекомендуется) или "Superseded" (отменено). Неудачные предложения принимаются как издержки процесса, и не считается зазорным быть связанным с таким предложением.

Управляющий комитет IETF (IESG или Internet Engineering Steering Group — группа технического управления Internet) отвечает за продвижение успешных RFC-документов в виде стандартов путем присвоения им грифа "Proposed Standard" (предложенный стандарт). Для того чтобы RFC квалифицировался как стандарт, спецификация должна быть стабильной, пройти экспертную оценку и привлечь значительный интерес Internet-сообщества. Опыт реализации не является абсолютно необходимым, прежде чем RFC получит обозначение Proposed Standard, но считается весьма желательным, и IESG может его потребовать, если RFC затрагивает основные протоколы или иным образом способен дестабилизировать Internet.

Предложенные стандарты постоянно пересматриваются и даже могут быть отозваны, если IESG и IETF найдут лучшее решение. Их не рекомендуется использовать в "средах, чувствительных к нарушениям", поэтому не следует применять их в системах управления

авиатранспортом или в оборудовании для интенсивной терапии.

Когда существует как минимум две работающие, полные, независимо созданные и способные к взаимодействию реализации предложенного стандарта, IESG может повысить стандарт до статуса "Draft Standard" (проект стандарта). RFC 2026 гласит: "Повышение до проекта стандарта является главным повышением статуса и подтверждает уверенность в том, что данная спецификация сформировалась и будет полезной".

RFC-документ, получивший статус Draft Standard, будет изменяться только в целях исправления ошибок логики спецификации. Ожидается, что спецификации, имеющие такой статус, готовы для внедрения в чувствительные к нарушениям среды.

Когда проект стандарта прошел тест широко распространенной реализации и получил общее одобрение, он может быть назван Internet Standard (Стандарт Internet). Такие стандарты сохраняют свои RFC-номера и, кроме того, получают номер STD-серии. На момент написания данной книги существовало более 3 тыс. RFC и только 60 STD-стандартов.

RFC, отклоняющиеся от схемы стандартизации, могут быть обозначены как Experimental (экспериментальные), Informational (информационные; шуточные RFC тоже получают такое обозначение) или Historic (имеющие историческое значение). Последнее обозначение применимо к вышедшим из употребления стандартам. Цитата из RFC 2026: "Борцы за чистоту литературного языка предложили использовать слово Historical (исторически сложившийся); однако в данном случае "исторически сложившимся" является использование слова Historic."

Процесс создания стандартов IETF направлен на поощрение стандартизации, вызванной скорее практикой, чем теорией, и гарантирует, что стандартные протоколы подверглись скрупулезной экспертной оценке и тестированию. Успех данной модели подтверждается ее результатами — всемирной сетью Internet.

17.4. Спецификации — ДНК, код — РНК

Даже в давние времена PDP-7 Unix-программисты всегда (больше чем их коллеги, работающие с другими системами) были склонны рассматривать старый код как непригодный к повторному использованию. Это, несомненно, было продуктом Unix-традиции, которая акцентировала внимание на модульности, облегчающей выбраковку и замену небольших блоков систем без потери целого. Unix-программисты уяснили из опыта, что попытки сохранения плохого кода или плохого дизайна часто являются более трудоемкими, чем создание проекта заново. Там, где в других культурах программирования инстинкт подсказывал бы исправлять неповоротливые монолиты, потому что в них вложено много труда, инстинкт Unix-программиста обычно ведет к выбрасыванию старого кода и созданию нового.

Традиция IETF усилила этот инстинкт, научив нас считать код вторичным по отношению к стандартам. Стандарты являются тем, что позволяет программам взаимодействовать; они связывают технологии в единое целое. Опыт IETF показывает, что тщательная разработка стандартов, стремящаяся к сбору лучшего из существующей практики, является той формой сдержанности, которая достигает гораздо большего, чем претенциозные попытки переделать мир по образу нереализуемого идеала.

После 1980 года влияние данного урока стало более заметно в Unix-сообществе. Поэтому, несмотря на то, что стандарт ANSI/ISO C 1989 года не был абсолютно безупречным, он был

исключительно четким и практичным для стандарта такого размера и важности. Единая спецификация Unix содержит "атавизмы" трех десятилетий экспериментов и фальстартов в более сложной области и, следовательно, является более беспорядочной, чем ANSI C. Однако компонентные стандарты, из которых она собрана, весьма хороши; твердым доказательством этого является тот факт, что, читая ее, Линус Торвальдс успешно создал Unix с нуля. Скромный, но мощный пример IETF создал один из критически важных блоков среды, которая сделала возможным подвиг Торвальдса.

Уважение к опубликованным стандартам и процессу IETF глубоко проникло в Unix-культуру. Умышленное нарушение Internet STD-стандартов просто невыполнимо. Это может иногда создавать пропасть взаимного непонимания между людьми с Unix-опытом и другими, склонными предполагать, что наиболее популярная или широко распространенная реализация протокола по определению является верной — даже если она нарушает стандарт так жестко, что не способна взаимодействовать с тщательно согласованным программным обеспечением.

Для Unix-программиста характерно уважение к опубликованным стандартам, поэтому он, скорее всего, будет враждебно относиться к спецификациям других видов. К тому времени, когда "водопадная модель" (исчерпывающая спецификация, затем реализация, затем отладка без возврата к любой стадии) утратила популярность в литературе по программной инженерии, она уже в течение многих лет была предметом насмешек среди Unix-программистов. Опыт и сильная традиция совместной разработки уже научили их, что лучший способ заключается в создании прототипов и повторных циклах тестирования и развития.

Традиция Unix четко определяет, что спецификации могут иметь большую ценность, но требует, чтобы они интерпретировались как временные и постоянно пересматривались в процессе практического использования так же, как пересматриваются стандарты Internet-Drafts и Proposed Standards. В лучшей практике Unix документация на программу используется как спецификация, подлежащая пересмотру, аналогично предложенному стандарту Internet.

В отличие от других сред, в Unix-разработке документация часто пишется до программы или, по крайней мере, совместно с ней. Для X11 основные стандарты X были закончены до выхода первой версии системы X и с тех пор остались по существу неизменными. Совместимость между различными X-системами совершенствуется далее путем скрупулезного тестирования, управляемого спецификацией.

Наличие хорошо написанной спецификации значительно упростило разработку тестового комплекта для системы X. Каждое положение в X-спецификации было преобразовано в код для проверки реализации. В ходе данного процесса в спецификации было обнаружено несколько незначительных несоответствий, но результатом является тестовый комплект, который охватывает значительную часть кодовых путей внутри пробной X-библиотеки и сервера, и ни один из них не ссылается на исходный код данной реализации. Кит Паккард.

Частичная автоматизация создания тестовых комплектов обоснованно считается главным преимуществом. Практический опыт и достижения в области графического искусства побудили многих критиковать систему X относительно конструкции, а различные ее части (такие как безопасность и модели пользовательских ресурсов) кажутся неповоротливыми и слишком усложненными. Но несмотря на это реализация X достигла выдающегося уровня стабильности и совместимости систем, создаваемых различными поставщиками.

В главе 9 обсуждалось значение переноса программирования на как можно более высокий уровень в целях минимизации влияния постоянной плотности ошибок. Цитата Кита Паккарда скрывает в себе идею о том, что документация системы X представляет собой не просто

перечень пожеланий, а некоторую форму высокоуровневого кода. Другой ведущий разработчик X подтверждает это мнение.

В X-системах спецификация всегда превалировала. Иногда спецификации содержат ошибки, которые также необходимо исправлять, но обычно ошибки встречаются чаще в коде, чем в спецификации (во всяком случае, это характерно для любой стоящей спецификации). Джим Геттис.

Далее Джим отмечает, что процесс развития X фактически весьма похож на процесс IETF. Его польза не ограничивается конструированием хороших тестовых комплектов; это означает, что спорить о поведении системы можно на функциональном уровне по отношению к спецификации, предотвращая слишком большую путаницу в вопросах реализации.

Продуманная разработка, определяемая спецификацией, допускает небольшую дискуссию об ошибках в сравнении с функциями; система, которая некорректно реализует спецификацию, неустойчива и должна быть исправлена.

Полагаю, эта идея настолько проникла в большинство из нас, что мы забываем о ее силе. Мой друг, работавший для небольшой программной фирмы восточнее Белвю, удивлялся, как разработчики Linux-приложений могут получать изменения операционной системы, синхронизированные с версиями приложений. В его компании главные системные API-интерфейсы часто изменялись, для того чтобы приспособиться к капризам приложений, и поэтому важнейшая функциональность системы часто должна была обновляться наряду с каждым приложением.

Я описал силу подчинения спецификациям и то, как от них зависит реализация, а затем перешел к утверждению, что приложение, которое получает неожиданный результат от документированного интерфейса, либо неверно спроектировано, либо содержит ошибку. Он нашел эту идею удивительной.

Распознавание таких ошибок является просто вопросом проверки реализации интерфейса относительно спецификации. Естественно, наличие исходного кода для данной реализации несколько упрощает это. Кит Паккард.

Позиция предшествующих стандартов имеет свои преимущества также и для конечных пользователей. Тогда как уже не маленькая компания восточнее Белвю имеет проблемы, сохраняя совместимость офисного набора с его предыдущими версиями, GUI-приложения, написанные в 1988 году, до сих пор работают без изменений на нынешних реализациях системы X. В мире Unix подобное долголетие является нормой, а причиной этого является позиция "стандарт как ДНК".

Таким образом, опыт показывает, что культура Unix, культивирующая уважение к стандартам, а также отбрасывание и переделку ошибочного кода, часто приводит к большей возможности взаимодействия в течение более продолжительного периода времени, чем постоянное исправление базы кода без стандарта, который обеспечивает управление и непрерывность. Это, несомненно, может быть одним из наиболее важных уроков Unix.

Последнее замечание Кита непосредственно приводит к вопросу, который выдвигается на передний план благодаря успеху Unix-систем с открытым исходным кодом — связь между открытыми стандартами и открытым исходным кодом. Данная проблема рассматривается в конце главы, но перед этим необходимо изучить практический вопрос: каким образом Unix-программисты могут действительно

использовать огромную массу накопленных стандартов и как овладеть практическими знаниями для достижения переносимости программного обеспечения?

17.5. Программирование, обеспечивающее переносимость

Переносимость программ обычно рассматривается в квази-пространственных понятиях: возможно ли перенести данный код на существующее аппаратно-программное обеспечение, отличное от того, для которого код создавался? Однако десятки лет опыта Unix подсказывают, что долговечность во времени важна в той же степени, если не в большей. Если бы можно было подробно предсказать будущее программного обеспечения, то оно, вероятно, было бы настоящим. Тем не менее, в программировании, обеспечивающем переносимость, следует пытаться обдумывать варианты построения программ на основе тех функций окружающей системы, которые, вероятнее всего, сохранятся, и избегать технологий, которые, скорее всего, в обозримом будущем прекратят свое существование.

Что касается Unix, два десятка лет изучения вопросов определения переносимых API-интерфейсов позволили полностью решить данную проблему. Средства, описанные в Единой спецификации Unix, вероятно, будут присутствовать во всех современных Unix-платформах и вряд ли не будут поддерживаться в будущем.

Однако не все зависимости от платформы разрешаются с помощью системных или библиотечных API-интерфейсов. Язык реализации также имеет значение; кроме того, проблемы могут быть обусловлены структурой файловой системы и конфигурационными отличиями между исходной и целевой системой. Однако практика Unix уже подтвердила способы решения данных проблем.

17.5.1. Переносимость и выбор языка

Первым вопросом в программировании, обеспечивающем переносимость, является выбор языка реализации. Все основные языки, рассмотренные в главе 14, являются высоко переносимыми в том смысле, что совместимые реализации доступны на всех современных Unix-системах, а для большинства из них также доступны реализации для Windows и MacOS. Проблемы переносимости часто возникают не в основных языках, а в библиотеках поддержки и степени интеграции с локальным окружением (особенно в IPC-методах и управлении параллельными процессами, включая инфраструктуру для GUI-интерфейсов).

17.5.1.1. Переносимость C

Базовый язык C в высшей степени переносим. Его стандартной реализацией в Unix является GNU C-компилятор, который повсеместно распространен не только Unix-системах с открытым исходным кодом, но и современных коммерческих вариантах операционной системы. GNU C был перенесен на Windows и классическую MacOS, но ни в одной из этих сред не стал широко распространенным, поскольку испытывает недостаток в переносимых привязках к собственным GUI-интерфейсам данных систем.

Стандартная библиотека ввода/вывода, математические подпрограммы и поддержка интернационализации переносимы во всех реализациях C. Файловый ввод/вывод, сигналы и управление процессами являются переносимыми между вариантами Unix, при условии, что

используются только современные API-интерфейсы, описанные в Единой спецификации Unix; более старый C-код часто содержит нагромождения условных операторов препроцессора, направленных на достижение переносимости, но они поддерживают интерфейсы, появившиеся до стандарта POSIX, из старых коммерческих Unix-систем, которые по состоянию на 2003 год устарели или близки к этому.

Более серьезные проблемы с переносимостью C начинаются в области IPC, параллельных процессов и GUI-интерфейсов. Вопросы переносимости IPC и параллельных процессов обсуждались в главе 7. Реальную практическую проблему создают GUI-инструментарии. Большое количество GUI-инструментариев с открытым исходным кодом универсально переносятся между современными вариантами Unix, а также в Windows и классическую MacOS — Tk, wxWindows, GTK и Qt — четыре хорошо известных вида инструментария с открытым исходным кодом и документацией, которую нетрудной найти в Web. Однако ни один из них не распространяется со всеми платформами, и (по причинам более юридическим, чем техническим) ни один из них не предоставляет на всех платформах естественный вид и восприятие GUI-интерфейса. В главе 15 даны некоторые рекомендации, позволяющие справиться с данной проблемой.

По теме разработки переносимого C-кода написаны целые тома. Эта книга не входит в их число, и авторы рекомендуют внимательно изучить

"Recommended C Style and Coding Standards" [11], а также главу о переносимости из книги

"The Practice of Programming" [40].

17.5.1.2. Переносимость C++

Для C++ на уровне операционной системы характерны те же проблемы переносимости, что и для C, а также ряд собственных. Одной из дополнительных проблем является то, что GNU компилятор с открытым исходным кодом для C++ значительно отстает от коммерческих реализаций; поэтому к середине 2003 года не существовало универсально внедряемого эквивалента GNU C, на котором можно основывать стандарт де-факто. Более того, ни один из компиляторов C++ не реализовывает полностью ISO-стандарт C++99, хотя GNU C++ подошел к этому очень близко.

17.5.1.3. Переносимость shell

Переносимость shell-сценариев, к сожалению, является низкой. Проблема заключается не в самой оболочке;

bash(1) (Bourne Again shell с открытым исходным кодом) распространена достаточно широко, для того чтобы малоразвитые shell-сценарии могли выполняться почти в любой среде. Проблема заключается в том, что в большинстве shell-сценариев интенсивно используются другие команды и фильтры, которые являются менее переносимыми, и их присутствие на какой-либо определенной целевой машине никоим образом не гарантируется.

Данную проблему можно преодолеть героическими усилиями, как в инструментах

autoconf(1) . Однако это действительно достаточно трудно, и большинство сложнейших

случаев программирования, которые обычно реализовались в shell, переместились к языкам сценариев второго поколения, таким как Perl, Python и Tcl.

17.5.1.4. Переносимость Perl

Perl отличается хорошей переносимостью. В стандартном варианте языка даже предоставляется переносимый набор привязок к Tk-инструментарию, который поддерживает переносимые GUI-интерфейсы в Unix, MacOS и Windows. Однако этому мешает одна проблема. Perl-сценарии часто требуют добавочных библиотек из архива CPAN (Comprehensive Perl Archive Network — полный сетевой архив Perl), которые не обязательно присутствуют в любой реализации Perl.

17.5.1.5. Переносимость Python

Python имеет превосходную переносимость. Как и Perl, стандартный вариант Python предоставляет переносимый набор привязок к Tk-инструментарию, который поддерживает переносимые GUI-интерфейсы в Unix, MacOS и Windows.

Стандартный Python обладает более развитой стандартной библиотекой, чем Perl и не имеет эквивалента архиву CPAN, на который могли бы полагаться программисты. Вместо этого важные модули расширения регулярно встраиваются в стандартный дистрибутив Python в ходе выпуска второстепенных версий. Это заменяет пространственную проблему временной — Python менее подвержен влиянию эффекта недостающих модулей. За это приходится расплачиваться тем, что второстепенные версии Python отчасти более важны, чем главные версии Perl. На практике этот компромисс благоприятствует Python.

17.5.1.6. Переносимость Tcl

Tcl демонстрирует хорошую переносимость в целом, но она сильно различается в зависимости от сложности проекта. Tk-инструментарий для кроссплатформенного GUI-программирования является естественным для Tcl. Как и в случае с Python, развитие основного языка проходит относительно гладко с немногими проблемами перекоса версий. К сожалению, Tcl даже более, чем Perl, зависит от средств расширения, которые не обязательно поставляются с каждой реализацией языка, а кроме того, не существует эквивалента CPAN-архива для централизованного их распространения.

Таким образом, для мелких проектов, не зависящих от расширений, переносимость Tcl превосходна. Однако крупные проекты часто сильно зависят как от расширений, так и (как в случае с shell-программированием) от вызываемых внешних команд, которые могут отсутствовать на целевой машине; часто они имеют слабую переносимость.

Парадоксально, но Tcl может страдать от простоты добавления в него расширений. К тому моменту, когда определенное расширение становится интересным в качестве части стандартного дистрибутива, существует, как правило, несколько различных его версий. На симпозиуме разработчиков Tcl/Tk (Tcl/Tk Workshop 1995 года) Джон Аустерхоут (John

Ousterhout) объяснил, почему в стандартном дистрибутиве Tcl отсутствует поддержка ОО-средств, так:

Представьте себе пять мулл, сидящих неподалеку, каждый из которых говорит: "Убей его, он неверный". Если бы я внедрил специфическую ОО-схему в ядро, то один из них сказал бы: "Благословляю тебя, сын мой, ты можешь поцеловать мой перстень", а остальные четыре сказали бы "Убей его, он неверный".

17.5.1.7. Переносимость Java

Переносимость Java превосходна — в конце концов, основной целью создания языка был девиз "написанное однажды работает везде". Вместе с тем переносимость Java не идеальна. Трудности в основном связаны с проблемами перекоса версий между JDK 1.1 и более старым GUI-инструментарием AWT (с одной стороны) и JDK 1.2 и более новым Swing. Это обусловлено несколькими важными причинами.

• Конструкция AWT Sun была настолько неадекватной, что ее необходимо было заменить инструментарием Swing.

• Отказ Microsoft от поддержки Java-разработки на Windows и попытка заменить данный язык C#.

• Решение Microsoft удерживать поддержку аплетов в Internet Explorer на уровне JDK 1.1.

• Лицензионные положения Sun, которые делают невозможной реализацию JDK 1.2 с открытым исходным кодом, замедляя внедрение пакета (особенно в мире Linux).

Разрабатывая программы, в которых задействованы GUI-интерфейсы, Java-разработчики, ищущие переносимости, в обозримом будущем окажутся перед лицом выбора: для сохранения максимальной переносимости (включая Microsoft Windows) остановиться на JDK1.1/AWT со слабо спроектированным инструментарием или получить лучший инструментарий и средства JDK 1.2, жертвуя некоторой переносимостью.

Наконец, как отмечалось выше, поддержка параллельных процессов в Java имеет проблемы переносимости. В отличие от менее претенциозных привязок к операционной системе для других языков, Java API действительно мог послужить в качестве моста между расходящимися моделями процессов, предоставляемыми различными операционными системами. Но это не решает проблему в полной мере.

17.5.1.8. Переносимость Emacs Lisp

Emacs Lisp отлично переносится на разные платформы. Инсталляции Emacs обновляются часто, поэтому действительно устаревшие среды встречаются редко. Lisp одного расширения поддерживается везде, и фактически все расширения поставляются с самим Emacs.

Кроме того, также весьма стабилен набор примитивов Emacs. Он достиг завершенности для задач, которые в течение многих лет возлагались на редактор (манипуляция буферами, обработка текста). Только введение системы X нарушило эту картину, и очень немногие режимы Emacs должны иметь сведения об X. Проблемы переносимости обычно являются

проявлением капризов средств операционной системы на С-уровне привязок; управление подчиненными процессами в таких режимах в качестве почтовых агентов — почти единственная область, где такие проблемы проявляются с непредсказуемой частотой.

17.5.2. Обход системных зависимостей

После выбора языка и библиотек поддержки следующим вопросом переносимости обычно является расположение ключевых системных файлов и каталогов: почтовых спулов, каталогов журнальных файлов и т.д. Прообразом данного типа проблем является то, где расположен каталог почтового спула — /var/spool/mail или /var/mail.

Часто можно избежать этого вида зависимости, отступив и воссоздав проблему. Зачем, так или иначе, открывать файл в каталоге почтового спула? Если запись в него необходима, то, возможно, лучшим решением будет просто вызов локального почтового транспортного клиента, с тем чтобы блокировка файла была выполнена верно. Если выполняется чтение из почтового каталога, то, вероятно, лучше будет обратиться к нему посредством POP3- или IMAP-сервера.

Подобные вопросы применимы и к другим системным файлам. Не лучше ли, например, вместо ручного открытия журнальных файлов воспользоваться средствами

syslog(3) . Интерфейсы вызова функций через библиотеку С стандартизированы лучше, чем расположение системных файлов, и этим следует пользоваться.

Если параметры расположения системных файлов должны присутствовать в коде, то наилучшая альтернатива зависит от способа распространения дистрибутива: исходный код или бинарная форма. Если распространяется исходный код, то могут помочь инструменты

autoconf, которые рассматриваются в следующем разделе. В случае распространения бинарных файлов хорошей практикой будет заставить программу получать начальные параметры системы и выяснять, возможно ли автоматически настроиться на локальные условия, например, путем проверки существования каталогов /var/mail и /var/spool/mail.

17.5.3. Инструменты, обеспечивающие переносимость

Часто для разрешения вопросов переносимости, обследования системной конфигурации и настройки make-файлов можно использовать GNU-утилиту с открытым исходным кодом

autoconf(1) , которая рассматривалась в главе 15. Пользователи, которые сегодня предпочитают компилировать программы из исходного кода, рассчитывают на возможность ввести команды configure; make; make install и получить чистую сборку. На странице <http://seul.org/docs/autotut/> публикуется хорошее учебное пособие по использованию данных инструментов. Даже если программа распространяется в бинарном виде, инструменты

autoconf(1) способны помочь автоматизировать решение проблемы подстройки кода под условия различных платформ.

Существуют другие инструменты, которые решают данную проблему. Двумя из наиболее

широко известных средств являются утилита

Imake(1) , связанная с системой X Window и инструмент Configure, созданный Ларри Уоллом (Larry Wall, создавшим позднее язык Perl) и приспособленный для многих различных проектов. Все они, как минимум, также сложны как набор autoconf, и используются в настоящее время не чаще. Они не охватывают такой же широкий диапазон целевых систем как

autoconf .

17.6. Интернационализация

Детальное рассмотрение интернационализации кода — разработки программы так, чтобы ее интерфейс легко вмещал в себя несколько языков и справлялся с причудами различных наборов символов — выходит за рамки тематики данной книги. Однако опыт Unix предлагает несколько уроков хорошей практики.

Во-первых, рекомендуется

разделять базу сообщений и код . Хорошая Unix-практика заключается в отделении строк сообщений, которые использует программа, от ее кода так, чтобы словари сообщений на других языках можно было подключать без модификации кода.

Наиболее известным инструментом для решения данной задачи является GNU-утилита

gettext , которая требует заключать в специальный макрос строки исходного языка, которые необходимо интернационализировать. Макрос использует каждую строку как ключ в словарях всех языков, которые могут поставляться в виде отдельных файлов. Если такие словари не доступны (или доступны, но в процессе поиска соответствие не было найдено), то макрос просто возвращает свой аргумент, безоговорочно отступая к собственному языку в коде.

Несмотря на то, что сама по себе программа

gettext к середине 2003 года являлась хрупкой и беспорядочно организованной, ее общая философия вполне логична. Для многих проектов вполне возможно создать легковесную версию данной идеи с хорошими результатами.

Во-вторых, в современных Unix-системах прослеживается отчетливая тенденция избавляться от всего "исторического мусора", связанного с множеством таблиц символов, и делать естественным языком приложений UTF-8, 8-битовое кодирование Unicode-символов со смещением (в противоположность, например, использованию в качестве такого языка 16-битовые широкие символы). Нижние 128 символов UTF-8 представляют собой символы ASCII, а нижние 256 символов — Latin-1, что означает обратную совместимость с двумя наиболее широко используемыми таблицами символов. Этому способствует тот факт, что XML и Java сделали данный выбор, но движущая сила уже определена, даже если бы не было XML и Java.

В-третьих, необходимо внимательно относиться к диапазонам символов в регулярных выражениях. Элемент [a-z] не обязательно охватывает все строчные буквы, если сценарий или программа разрабатывается, например, для Германии, где острая буква "s" или символ "ss" считается строчным, но не входит в данный диапазон. Подобные проблемы возникают с французскими буквами под ударением. Надежнее использовать элемент [[:lower:]] и другие символьные диапазоны, описанные в стандарте POSIX.

17.7. Переносимость, открытые стандарты и открытый исходный код

Переносимость требует стандартов. Эталонные реализации с открытым исходным кодом являются наиболее эффективным из известных методов, как для распространения стандарта, так и для того, чтобы вынудить коммерческих поставщиков согласиться со стандартом. Для разработчика реализация опубликованного стандарта в открытом исходном коде может одновременно сокращать объем работ по кодированию и позволяет создаваемому продукту извлечь преимущества (как ожидаемые, так и неожиданные) из труда других разработчиков.

Рассмотрим, например, разработку программы захвата изображений для цифровой камеры. Зачем создавать собственный формат для сохранения изображений или покупать коммерческий код, когда (как было сказано в главе 5) существует многократно проверенная, полнофункциональная библиотека для записи PNG-изображений, реализованная в открытом исходном коде?

Создание (или воссоздание) открытого исходного кода также оказало значительное влияние на процесс стандартизации. Несмотря на то, что это не является официальным требованием, IETF приблизительно с 1997 года все больше сопротивляется превращать в стандарты документы RFC, которые не имеют по крайней мере одной эталонной реализации с открытым исходным кодом. В будущем кажется вполне вероятным, что степень соответствия любому определенному стандарту будет все больше измеряться соответствием реализациям с открытым исходным кодом, которые были одобрены авторами стандарта.

Оборотной стороной этого является то, что часто наилучший способ сделать что-нибудь стандартом заключается в распространении высококачественной реализации с открытым исходным кодом. Генри Спенсер.

В завершение следует подчеркнуть, что самый эффективный шаг, который можно предпринять для обеспечения переносимости разрабатываемого кода заключается в том, чтобы не полагаться на частную технологию. Не известно, когда закрытая библиотека, инструмент, генератор кода или сетевой протокол, от которого зависит программа, прекратит существование, или, когда интерфейс будет изменен каким-либо обратно несовместимым путем, который разрушит разрабатываемый проект. Используя открытый исходный код, можно обеспечить проекту будущее, даже если ведущая версия изменится так, что разрушит проект; ввиду того, что существует доступ к исходному коду, можно будет в случае необходимости перенести его на новую платформу.

До конца 90-х годов данная рекомендация была бы невыполнимой. Несколько альтернатив частных операционных систем и средств разработки были "благородными экспериментами", подтверждениями академических идей или "игрушками". Но Internet изменил все. В середине 2003 года Linux и другие Unix-системы с открытым исходным кодом доказали свою стойкость как платформы для создания программного обеспечения промышленного качества. В настоящее время разработчики имеют лучшую альтернативу, чем зависимость от краткосрочных бизнес- решений, направленных на защиту чужой монополии. Применяйте защищенные конструкции — создавайте программы на основе открытого исходного кода и вы не попадете в безвыходное положение.

18

Я никогда не встречал человека, которому хотелось бы прочесть 17 000 страниц документации, а если бы такой человек был, я убил бы его для очистки генофонда. —Джозеф Костелло (Josef Costello)

Первым применением Unix в 1971 году была платформа для компоновки документов — Bell Labs использовала ее для подготовки патентных документов в целях составления картотеки. Фотонабор, управляемый компьютером, все еще оставался тогда новой идеей, и на несколько лет после своего дебюта в 1973 году форматер, который создал Джо Оссана (Joe Ossana),

troff(1), определил уровень техники.

С тех пор сложные форматеры документов, программное обеспечение для набора, а также различные программы макетирования страниц занимают важное место в традициях Unix. Хотя

troff(1) доказал свою удивительную долговечность, в Unix также поддерживается значительная работа в данной прикладной области. Сегодня Unix-разработчики и Unix-инструменты находятся на передовом рубеже далеко идущих перемен в практике создания документов, началом которой послужило появление World Wide Web.

На уровне пользовательских представлений с середины 1990-х годов практика сообщества Unix быстро смещается в направлении идеи "все документы — HTML, все ссылки — URL". Более того, современные программы для просмотра справочной информации в Unix являются просто Web-браузерами, которые способны осуществлять синтаксический анализ определенных видов URL-адресов (например, URL "man:ls(1)" интерпретирует man-страницу

ls(1) в HTML-страницу). Это облегчает проблемы, вызванные существованием множества различных форматов для главных документов, но не решает их полностью. Составители документов до сих пор вынуждены решать проблему выбора главного формата, который удовлетворит их специфические потребности.

В данной главе рассматривается весьма неудачное разнообразие различных форматов документов и инструментов, которые остались позади за десятилетия экспериментов. Кроме того, в этой главе формулируются несколько определяющих правил хорошей практики и удачного стиля.

18.1. Концепции документации

В первую очередь необходимо определить различия между WYSIWYG-программами (WYSIWYG, "What You See Is What You Get" — "что видишь, то и получаешь") и

инструментами разметки (Markup-Centered Tools) . Большинство настольных издательских программ и текстовых процессоров входят в первую категорию; они имеют GUI-интерфейсы, в которых вводимый пользователем текст вставляется непосредственно в экранное представление документа, которое задумано как наиболее точное подобие окончательной печатаемой версии. Напротив, в системе, ориентированной на разметку, документ обычно

является простым текстовым документом, содержащим явные, видимые управляющие теги и вообще не имеет сходства с ожидаемой выходной версией. Размеченный исходный документ можно модифицировать с помощью обычного текстового редактора, но необходимо передать программе-форматеру, создающей визуализированный вывод для печати или отображения на экране.

Визуальный интерфейс, или WYSIWYG-стиль, был чрезмерно затратным для ранних компьютеров и использовался редко вплоть до появления в 1984 году персонального компьютера Macintosh. В настоящее время этот стиль полностью доминирует в не-Unix операционных системах. С другой стороны, собственные инструменты форматирования документов в Unix почти все основаны на разметке. Unix-программа troff(1) 1971 года была форматером разметки и, вероятно, является старейшей из подобных программ, которая используется до сих пор.

Инструменты разметки до сих пор играют значительную роль, поскольку имеющиеся реализации WYSIWYG часто приводят к различным сбоям — некоторые из них поверхностные, а некоторые глубокие. WYSIWYG-процессоры имеют одну общую проблему с GUI-интерфейсами, которая рассматривалась в главе 11. Тот факт, что пользователь

может визуально манипулировать любым объектом, часто означает то, что пользователь

должен управлять всем. Данная проблема осталась бы, даже если WYSIWIG-соответствие между экранным выводом и распечаткой было бы идеальным, но это почти всегда не так.

В действительности WYSIWYG-процессоры не являются полностью WYSIWYG. Большинство из них имеют интерфейсы, скрывающие от пользователя различия между экранным представлением и выводом принтера, фактически не устраняя эти различия. Таким образом, нарушается правило наименьшей неожиданности: визуальный аспект подталкивает к использованию программы как печатной машинки, даже если она таковой не является, а входные данные время от времени приводят к неожиданному и нежелательному результату.

Верно также и то, что WYSIWIG-системы фактически опираются на код разметки, но "делают все возможное", для того чтобы он стал невидимым при обычном использовании программы. Следовательно, нарушается правило прозрачности: пользователю не видна вся разметка, поэтому трудно привести в порядок документы, которые испорчены в результате несоответствующей разметки.

Несмотря на свои проблемы, WYSIWYG-процессоры могут приносить пользу, если все потребности пользователя сводятся к перемещению картинки на обложке четырехстраничной брошюры вправо на три пункта. Однако они способны ограничивать пользователя всякий раз, когда необходимо внести глобальные изменения в макет 300-страничной рукописи. Пользователи WYSIWYG-процессоров, столкнувшиеся с данной проблемой, вынуждены сдаться или тысячи раз щелкать мышью. В подобных ситуациях действительно ничто не может заменить возможность редактирования явной разметки, и Unix-инструменты, ориентированные на разметку, предлагают лучшие решения.

Сегодня в мире, находящемся под влиянием Web и XML, становится общепринятым разграничивать разметку

представления и

структурную разметку в документах. Первую составляют инструкции о том, как должен выглядеть документ, а вторую — инструкции о том, как он организован и что означает. Такое разграничение не было очевидно понятным или соблюдаемым в ранних Unix-инструментах, однако оно важно для понимания вынуждающих факторов проектирования, которые привели к их современным потомкам.

Разметка уровня представления хранит всю информацию для форматирования (например, о желаемом расположении пустых мест и изменении шрифтов) в самом документе. В системах структурной разметки документ должен быть соединен с

таблицей стилей (stylesheet) , которая дает форматеру указания о том, как преобразовывать структурную разметку документа в физическую компоновку. Оба вида разметки, в конечном итоге, управляют внешним видом печатаемого или просматриваемого документа, однако структурная разметка делает это посредством еще одного уровня преобразования, которое оказывается необходимым, если требуется обеспечить хорошие результаты, как для печати документа, так и для его представления в Web-среде.

Большинство систем разметки документов поддерживают макросредства. Макросы являются определяемыми пользователями командами, которые с помощью подстановки текста разворачиваются во встроенные запросы разметки. Обычно данные макросы добавляют в язык разметки структурные функции (такие как возможность объявлять заголовки разделов).

Наборы макросов troff (mm, me и мой пакет ms) были фактически предназначены для того, чтобы увести пользователей от редактирования, ориентированного на форматирование, в направлении редактирования, ориентированного на содержание документов. Идея заключалась в отметке семантических частей и в последующем использовании различных стилевых пакетов, которые "знали бы", следует ли выделять в данном стиле заголовок жирным шрифтом или нет, центрировать его или нет, и так далее. Таким образом, в одной точке был набор макросов, которые пытались имитировать ACM-стиль, и другой набор, имитировавший стиль Physical Review, но использующий базовую разметку -ms. Все эти макросы не имели успеха у людей, которые сосредотачивались на создании одного документа и управлении его внешним видом, так же как Web-страницы "тонут" в дискуссиях о том, кто должен управлять внешним видом, — автор или читатель. Я часто встречал секретарей, которые использовали команду .AU (author name — имя автора) только для того, чтобы выделить текст курсивом, заметив, что она выполняет это действие, а затем сталкивались с трудностями при использовании других эффектов команды. Майк Леск.

Наконец, следует отметить, что существует значительные различия между тем, что составители хотят сделать с небольшими документами (деловыми и личными письмами, брошюрами, бюллетенями), и тем, что они хотят делать с "крупными" документами (книгами, длинными статьями, техническими докладами и руководствами). В таких документах часто имеется более развитая структура, они часто составляются из частей, которые, возможно, требуется редактировать по отдельности, а кроме того, они нуждаются в автоматически генерируемых возможностях, таких как оглавления. Все это характерные особенности, которые поддерживают инструменты, ориентированные на разметку.

18.2. Стиль Unix

Стиль документации в Unix (и инструментов для подготовки документов) имеет несколько технических и культурных особенностей, которые обособляют его от способа создания документации в других системах. Понимание данных характерных особенностей, во-первых, создаст основу для понимания того, почему программы и практические приемы выглядят именно так, и во-вторых, почему документация читается именно таким способом.

18.2.1. Склонность к большим документам

Средства подготовки документов в Unix всегда были предназначены, главным образом, для разрешения трудностей, связанных с компоновкой крупных и сложных документов. Первоначально такими документами были патентные заявки и техническая документация. Позднее — научные и технические статьи и техническая документация всех видов. В результате большинство Unix-разработчиков научились ценить средства разметки документов. В отличие от PC-пользователей того времени, Unix-культура не была под впечатлением WYSIWYG текстовых процессоров, когда они стали широко доступными в конце 80-х годов и в начале 90-х годов прошлого века. Даже среди молодых современных Unix-хакеров достаточно необычно встретить того, кто действительно отдает предпочтение WYSIWYG-системам.

Неприятие трудных для понимания двоичных форматов документов и особенно непонятных

частных двоичных форматов также сыграло свою роль в отклонении WYSIWYG-инструментов. С другой стороны, Unix-программисты с энтузиазмом ухватились за PostScript (современный стандарт языка для управления графическими принтерами) как только документация на язык стала доступной. Данный стандарт точно вписывается в Unix-традиции узкоспециальных языков. Современные Unix-системы с открытым исходным кодом имеют превосходные инструменты для работы с форматами PostScript и PDF (Portable Document Format).

Другим последствием этой истории является то, что Unix-инструменты для работы с документами часто имеют сравнительно слабую поддержку включения изображений, но хорошо поддерживают диаграммы, таблицы, графики и математические формулы, т.е. то, что часто требуется в технических документах.

Привязанность Unix к системам разметки часто высмеивается как предубеждение или характерная черта замкнутых хакеров, но не является ни тем, ни другим. Так же как считающийся "примитивным" CLI-стиль, Unix во многих случаях лучше, чем GUI-интерфейсы, адаптируется к потребностям опытных пользователей, ориентированная на разметку конструкция таких инструментов, как

troff(1) , подходит для потребностей опытных компоновщиков документов лучше, чем WYSIWYG-программы.

Уклон традиции Unix в сторону больших документов не только привязывает разработчиков к форматерам на основе разметки, подобным

troff , но также и заинтересовывает их в использовании структурной разметки. История развития документирующих средств Unix является историей прерывистого, запутанного и непостоянного движения в общем направлении от разметки уровня представления к структурной разметке. В середине 2003 года данная миграция все еще не была закончена, но ее конец близок

Развитие World Wide Web означало, что способность отображать документы в различных средах (или, по крайней мере, печатать и отображать HTML-документы) стало после 1993 года главной задачей для инструментов документирования. В то же время даже обычные пользователи под влиянием HTML более комфортно стали воспринимать системы, ориентированные на разметку. Это привело непосредственно к росту заинтересованности в структурной разметке и изобретению XML после 1996 года. Внезапно давняя склонность Unix к системам разметки стала восприниматься скорее как перспективная, нежели реакционная.

Сегодня для Unix характерна наиболее передовая разработка основанных на XML средствах документирования с использованием структурной разметки. Но в то же время Unix-культура

еще должна освободиться от прежней традиции систем разметки уровня представления. "Скрипучий, лязгающий, бронированный динозавр", каковым является

troff , только частично был вытеснен языками HTML и XML.

18.2.2. Культурный стиль

Большая часть документации по программному обеспечению написана "техническими писателями", стремящимися найти наименьший общий знаменатель незнания — "умное написание для неосведомленных". Документация, которая поставляется с Unix-системами, традиционно пишется программистами для коллег. Даже если документация не написана программистами для программистов, то она часто находится под влиянием стиля и формата огромной массы такой документации, поставляемой с Unix-системами.

Данные различия можно объединить одним замечанием: man-страницы в Unix обычно содержат раздел, который называется "BUGS" (ошибки). В других культурах "технические писатели" пытаются придать продукту лучший вид, минуя известные ошибки или касаясь их вскользь. В культуре операционной системы Unix разработчики беспощадно описывают известные недостатки своего программного обеспечения, а пользователи рассматривают краткий, но информативный раздел "BUGS" как обнадеживающий знак качественной работы. Коммерческие дистрибутивы Unix, нарушившие данное соглашение, либо исключив раздел "BUGS", либо заменив его название более мягкими формулировками "LIMITATIONS" (ограничения) или "ISSUES" (проблемы), или "APPLICATION USAGE" (использование приложения), неизменно приходят в упадок.

Тогда как большая часть остальной программной документации часто колеблется между "непонятностью" и чрезмерно упрощенным, снисходительным изложением, классическая Unix-документация пишется в сжатом стиле, но вместе с тем является полной. Она "не ведет читателя за руку", а обычно указывает верное направление. Данный стиль предполагает активное чтение, при котором читатель способен сделать очевидный, но не высказанный вывод, на основе написанного и обладает достаточной уверенностью, чтобы доверять своим заключениям.

Unix-программисты склонны хорошо писать справочники, и большинство документации в Unix имеют вид справочников или

памяток для тех, кто думает подобно создателю документа, но еще не является экспертом в данном программном обеспечении. Результаты часто выглядят гораздо более загадочными и редкими, чем являются в действительности. Каждое слово документации необходимо читать очень внимательно, поскольку все, что необходимо знать, в ней, вероятно, имеется, а недостающие выводы можно сделать самостоятельно. Необходимо вдумчиво читать каждое слово, так как сведения редко повторяются дважды.

18.3. Многообразие форматов документации в Unix

Все основные форматы Unix-документации, кроме самых новых, являются разметками уровня представления, созданными с помощью макропакетов. Ниже они рассматриваются в порядке их возникновения.

18.3.1.

troff и инструментарий Documenter's Workbench

Структура и инструменты Documenter's Workbench рассматривались в главе 8 как пример интеграции системы, состоящей из нескольких мини-языков. Здесь рассматривается функциональная роль данного инструментария как системы форматирования документов.

Форматер

troff интерпретирует язык разметки уровня представления. Последние реализации данной программы, такие как GNU-проект

groff(1), по умолчанию на выходе создают PostScript-представление, хотя существует возможность получать другие формы вывода путем выбора подходящего драйвера. В примере 18.1 приводится несколько

troff- кодов, которые можно встретить в исходных текстах документов.

troff(1) имеет множество других запросов, но большинство из них вряд ли можно увидеть немедленно. Весьма немногие документы написаны на чистом

troff. Он поддерживает макросредства, и в нем более или менее широко используются около десятка макропакетов. Среди них наибольшее распространение получил макропакет

man(7) , который используется для написания man-страниц Unix. См. пример 18.2.

Две из почти десяти исторических библиотек макросов

troff, ms(7) и

mm(7) используются до сих пор. В BSD Unix имеется собственный замысловатый расширенный набор макросов,

mdoc(7) . Все они предназначены для написания технических руководств и длинных документов. Они имеют подобный стиль, но более сложны, чем man-макросы, и ориентированы на создание форматированного вывода.

Сокращенный вариант

troff(1) , который называется

nroff(1) , генерирует вывод для устройств, поддерживающих только моноширинные шрифты, например, для построчно-печатающих устройств и символьных терминалов. Отображаемая для пользователя внутри терминального окна man-страница Unix форматируется с помощью nroff.Пример 18.1. Разметка

groff(1)

Это непрерывный текст.

.\" Комментарии начинаются с обратной косой черты и двойных кавычек.

.ft B

Данный текст будет выделен жирным шрифтом.

.ft R

Данный текст будет выведен снова стандартным шрифтом (Roman).

Данные строки снова, как и "Это непрерывный текст", будут

отформатированы как заполненный абзац.

.bp

Запрос bp приводит к созданию новой страницы и разрыву абзаца.

Эта строка будет частью второго заполненного абзаца.

.sp 3

Запрос .sp выводит заданное аргументом количество пустых строк,

.nf

Запрос nf отключает заполнение абзаца.

До тех пор, пока оно снова не будет включено запросом fi,

пустые места и макет страницы будут сохраняться.

Одно слово данной строки будет выделено \fBbold\fR жирным шрифтом.

.fi

Снова включено заполнение абзаца.

Средства инструментария Documenter's Workbench очень хорошо решают задачи по форматированию технических документов, для которых они и предназначены, именно поэтому они непрерывно используются в течение более чем 30 лет, тогда как производительность компьютеров возросла в тысячи раз. Они создают форматированный текст приемлемого качества на графических принтерах и способны корректно отображать форматированные man-страницы на экране.

Однако в некоторых областях они плохо применимы. Их стандартный выбор доступных шрифтов ограничен. Они слабо обрабатывают изображения. Сложно обеспечить точное управление позиционированием текста, изображений или диаграмм внутри страницы. Поддержка многоязычных документов отсутствует. Существует множество других проблем. Некоторые из них хронические, но незначительные, другие абсолютно затрудняют специфическое применение. Но наиболее серьезная причина заключается в том, что, ввиду того, что большая часть разметки осуществляется на уровне представления, создавать хорошие Web-страницы из немодифицированных

troff- текстов очень трудно.

Тем не менее, на момент написания книги man-страницы остаются единственной наиболее важной формой Unix-документации. Пример 18.2. Разметка

man -страниц

.SH ПРИМЕР РАЗДЕЛА

Макрос SH начинает раздел, выделяя его заголовок жирным шрифтом.

.P

Запрос P начинает новый абзац. Запрос I форматирует

свой

аргумент

.I курсивом.

.IP *

Данный запрос начинает абзац с отступом, отмеченным звездочкой.

Дополнительный текст для первого абзаца списка.

.TP

Эта первая строка станет меткой абзаца

Данная строка будет первой строкой в абзаце, с отступом

по отношению к метке.

Пустая строка выше интерпретируется почти так же, как

разрыв абзаца (фактически как troff-запрос .sp 1).

.SS Подраздел

Это текст подраздела.

18.3.2. TEX

TEX (произносится как "теХ") представляет собой очень мощную программу подготовки документов, которая, подобно редактору Emacs, возникла за пределами Unix-культуры, но теперь "прижилась" в ней. Она была создана видным компьютерным ученым Дональдом Кнуттом (Donald Knuth), когда ему надоело качество доступной в конце 70-х годов прошлого века типографской разметки текста и особенно форматирования математических выражений.

TEX, подобно утилите

troff(1) , является системой, ориентированной на разметку. TEX имеет гораздо более мощный язык запросов, чем

troff . Среди прочего, данная программа лучше приспособлена для обработки изображений,

точного позиционирования содержимого страницы и интернационализации. особенно хорошо проявляет себя в форматировании математических выражений и является непревзойденным инструментом для решения базовых задач форматирования, таких как кернинг, заполнение строк и расстановка переносов. TEX стал стандартным форматом для представления материалов в большинство математических журналов. В настоящее время TEX курируется как программа с открытым исходным кодом рабочей группой Американского математического общества (American Mathematical Society). Программа также широко используется для создания научных документов.

Как и в случае с

troff(1) , пользователь обычно не пишет вручную множество чистых TEX-макросов. Вместо этого используются макро-пакеты и различные вспомогательные программы. Один особый макропакет, TEX является почти универсальным, и большинство пользователей, которые говорят о том, что создают документы в TEX, почти всегда фактически имеют в виду написание LATEX-макросов. Как и макропакеты

troff , множество запросов LATEX являются полуструктурными.

Одно важное применение TEX, которое обычно скрыто от пользователя, заключается в том, что другие инструменты обработки документов часто генерируют LATEX-код, преобразуемый впоследствии в PostScript, вместо того, чтобы самостоятельно пытаться выполнять гораздо более трудную работу по созданию PostScript-документа. Такой прием используется в управляющем сценарии

xmlto(1) , который рассматривался в главе 14 в качестве учебного примера shell-программирования. Аналогично данная проблема решается в инструментарии XML-DocBook, который описывается ниже в настоящей главе.

TEX имеет более широкий диапазон применений, чем

troff(1) , и в основном лучшую конструкцию. Во все более связанном с Web мире рассматриваемая программа имеет те же фундаментальные проблемы, что и

troff . Ее разметка сильно привязана к уровню представления, и автоматическое создание хороших Web-страниц из исходных текстов TEX затруднено и чревато сбоями.

Программа TEX никогда не используется для создания Unix-документации и только иногда используется для документирования приложений. Для таких целей достаточно использовать

troff . Однако некоторые программные пакеты, которые возникли в академических учреждениях за пределами Unix-сообщества, заимствовали использование TEX как главного формата документации; одним из примеров является язык Python. Как было сказано выше, он также интенсивно используется для создания математических и научных документов и, вероятно, будет доминировать в этой нише еще несколько лет.

18.3.3. Texinfo

Texinfo — разметка документации, разработанная Фондом свободного программного обеспечения. Она используется главным образом для документирования GNU-проектов, включая создание документации для таких важнейших инструментов, как Emacs и коллекция GNU-компиляторов.

Texinfo была первой системой разметки, специально предназначенной для поддержки вывода форматированного текста на бумагу и гипертекстового вывода для просмотра на экране. Однако используемым в ней гипертекстовым форматом был не HTML, а более примитивная разновидность, которая называется "info", первоначально предназначенная для просмотра из Emacs. Для печати документов Texinfo преобразуется в TEX-макрос, а затем из него в PostScript-представление.

Texinfo-инструменты в настоящее время способны генерировать HTML-документы. Однако качество и полнота данной операции оставляют желать лучшего, а поскольку множество Texinfo-разметки находится на уровне представления, вряд ли эта ситуация улучшится. В середине 2003 года Фонд свободного программного обеспечения еще продолжал работу над эвристическим преобразованием Texinfo в DocBook. Вероятно, формат Texinfo на какое-то время сохранится.

18.3.4. POD

POD или Plain Old Documentation (простая старая документация) представляет собой систему разметки, применяемую кураторами проекта Perl. Данная система генерирует страницы руководства и имеет все известные проблемы разметок уровня представления, включая проблемы при создании хороших HTML-документов.

18.3.5. HTML

С момента широкого распространения World Wide Web в начале 90-х годов прошлого века документация небольшой, но возрастающей части Unix-проектов пишется непосредственно на HTML. Проблема данного подхода заключается в том, что генерировать из HTML высококачественный типографский вывод трудно. Кроме того, существуют определенные проблемы с индексированием; необходимая для создания индексов информация в HTML отсутствует.

18.3.6. DocBook

DocBook представляет собой определение типов SGML- и XML-документов, предназначенное для крупных, сложных технических документов. Данный формат является единственным исключительно структурированным среди форматов разметки, используемых в Unix-сообществе. Инструмент

xmlto(1) , описанный в главе 14, поддерживает преобразование в HTML, XHTML, PostScript, PDF, разметку справочной системы Windows и несколько менее важных форматов.

Несколько главных проектов с открытым исходным кодом (включая Linux Documentation Project, FreeBSD, Apache, Samba, GNOME и KDE) уже используют DocBook в качестве главного формата. Данная книга была написана в XML-DocBook.

Изучение DocBook — большая тема. Ее рассмотрение будет продолжено после обобщения

проблем, связанных с современным состоянием подготовки документов в Unix.

18.4. Современный хаос и возможный выход из положения

В настоящее время Unix-документация является "неорганизованной смесью".

Главные файлы документации в современных Unix-системах разбросаны по восьми различным форматам разметки: man, ms, mm, TEX, Texinfo, POD, HTML и DocBook. Не существует унифицированного способа просматривать все сформированные версии. Они не доступны в Web и не снабжены перекрестными ссылками.

Многие представители Unix-сообщества осознают данную проблему. Во время написания данной книги большинство усилий, направленных на ее разрешение, исходит от разработчиков открытого исходного кода, которые более активно заинтересованы в конкуренции за предпочтения нетехнических конечных пользователей, чем разработчики частных Unix-систем. С 2000 года практика движется в направлении использования XML-DocBook как формата обмена документацией.

Ясная, но требующая больших усилий для достижения цель заключается в том, чтобы снабдить каждую Unix-систему программным обеспечением, которые будет функционировать как общесистемный реестр документации. Когда системные администраторы устанавливают пакеты, одним из этапов будет ввод в данный реестр XML-DocBook-документации для этих пакетов. Затем документация будет преобразована в общедоступное дерево HTML-документов и снабжена перекрестными ссылками с уже имеющейся документацией.

Ранние версии программ регистрации документов уже работают. Проблема преобразования других форматов в XML-DocBook является большой и запутанной, однако инструменты преобразования появляются в поле зрения. Другие политические и технические проблемы остаются нерешенными, однако их можно решить. Несмотря на то, что к середине 2003 года сообщество в целом не пришло к общему мнению о том, что более старые форматы необходимо постепенно прекратить использовать, это событие кажется самым вероятным.

Соответственно, далее формат DocBook и соответствующий инструментарий рассматривается более подробно. Данное описание следует читать как введение в XML в контексте Unix, практическое руководство для применения и как основной учебный пример. Оно является хорошим примером того, как в контексте сообщества Unix, вокруг общих стандартов развивается взаимодействие между различными проектными группами разработчиков.

18.5. DocBook

Большое количество главных проектов с открытым исходным кодом сходятся в том, что DocBook является стандартным форматом их документации. Сторонники разметки, основанной на XML, видимо, выиграли теоретический спор против разметки уровня представления и в пользу разметки структурного уровня, а эффективный XML-DocBook-инструментарий доступен в виде программ с открытым исходным кодом.

Тем не менее, DocBook и программы, поддерживающие данный формат, окружает большая неразбериха. Его приверженцы используют непонятный и неприступный даже с точки зрения

стандартов компьютерной науки жаргон, привязанный к аббревиатурам, которые не имеют очевидной связи с тем, что необходимо сделать для написания разметки и создания из нее HTML- или PostScript-представления. XML-стандарты и технические документы печально известны своей невразумительностью. В оставшейся части данного раздела предпринимается попытка прояснить жаргон.

18.5.1. Определения типов документов

(Необходимо отметить: для того чтобы сохранить простоту изложения, большая часть данного раздела содержит несколько искаженные сведения, т.е. опускается большая часть истории. Справедливость будет полностью восстановлена в последующем разделе.)

DocBook является языком разметки структурного уровня, а именно диалектом XML. DocBook-документ представляет собой блок XML-кода, в котором используются XML-теги для структурной разметки.

Форматеру для применения к документу таблицы стилей и придания ему соответствующего внешнего вида требуется сведения об общей структуре данного документа. Например, для того чтобы соответствующим образом физически отформатировать заголовки глав, необходимо сообщить форматеру о том, что рукопись книги обычно состоит из титульных элементов, последовательности глав и выходных данных. Для того чтобы сообщить эти сведения, необходимо предоставить форматеру

определение типа документа (Document Type Definition — DTD) . DTD сообщает форматеру о том, какие элементы могут содержаться в структуре документа и в каком порядке они могут появляться.

Называя DocBook "диалектом" XML, автор фактически имел в виду, что DocBook является DTD, причем довольно большим DTD, содержащим около 400 тегов[147].

В тени DocBook работает определенный вид программ, который называется

анализатором корректности (validating parser). Первым этапом при форматировании DocBook-документа является его обработка анализатором корректности (клиентская часть DocBook-форматера). Данная программа сравнивает документ с DocBook DTD, для того чтобы гарантировать, что пользователь не нарушает структурных правил DTD (в противном случае серверная часть форматера, т.е. та часть, которая применяет таблицу стилей, может быть поставлена в тупик).

Анализатор корректности либо перехватывает ошибку, отправляя пользователю сообщение о тех местах, где структура документа нарушена, либо транслирует документ в поток XML-элементов и текста, которые будут скомбинированы сервером с информацией из таблицы стилей для создания отформатированного вывода.

Данный процесс в целом схематически изображен на рис. 18.1.

Рис. 18.1. Обработка структурированных документов

Часть диаграммы внутри пунктирного блока соответствует форматирующему программному обеспечению или

инструментальной связке (toolchain) . Для того чтобы понять процесс, необходимо кроме очевидного и видимого ввода данных в форматер (исходного текста документа) учитывать

два вида скрытого ввода форматера (DTD и таблица стилей).

18.5.2. Другие DTD-определения

Небольшое отступление к другим DTD-определениям поможет прояснить, какие сведения предыдущего раздела являются специфичными для DocBook, а какие являются общими для всех языков структурированной разметки.

TEI (Text Encoding Initiative, <http://www.tei-c.org/>)— крупное, сложное DTD-определение, которое используется главным образом в академических учреждениях для компьютерной перезаписи литературных текстов. Применяемые в TEI связки Unix-инструментов состоят из множества тех же инструментов, которые задействованы в DocBook, но используются другие таблицы стилей и (естественно) другое DTD-определение.

XHTML, последняя версия HTML, также является XML-приложением, описанным DTD, что объясняет "родовое сходство" между XHTML- и DocBook-тегами. Инструментальная связка XHTML состоит из Web-браузеров, которые способны форматировать HTML как простой ASCII-текст, а также любого количества узкоспециальных утилит для печати HTML-текста.

Для того чтобы облегчить обмен структурированной информацией в других областях, таких как биоинформатика и банковское дело, поддерживаются также многие другие XML DTD-определения. Для того чтобы получить представление о доступном многообразии DTD-определений, можно воспользоваться списком репозитариев <http://www.xml.com/pub/rg/DTD_Repositories>.

18.5.3. Инструментальная связка DocBook

Обычно для создания XHTML-документа из исходного DocBook-текета используется интерфейсный сценарий

xmlto(1) . Ниже приводятся примерные команды.

bash$ xmlto xhtml foo.xml

bash$ ls *.html

ar01s02.html ar01s03.html ar01s04.html index.html

В данном примере XML-DocBook-документ с именем foo.xml, содержащий три раздела верхнего уровня, преобразовывается в индексную страницу и три части. Также просто создать одну большую страницу.

bash$ xmlto xhtml-nochunks foo.xml

bash$ ls *.html

foo.html

Наконец, ниже приводятся команды для создания PostScript-представления для печати.

bash$ xmlto ps foo.xml # Создание PostScript

bash$ ls *.ps

foo.ps

Для того чтобы превратить существующие документы в HTML или PostScript, необходимо ядро, которое способно применить к данным документам комбинацию DocBook DTD и подходящей таблицы стилей. На рис. 18.2 показано, как для данной цели совместно используются инструменты с открытым исходным кодом.

Рис. 18.2. Современная инструментальная связка XML-DocBook

Синтаксический анализ документа и его трансформация на основе таблицы стилей будет осуществляться одной из трех программ. Вероятнее всего, программой

xsltproc , которая представляет собой синтаксический анализатор, поставляемый с Red Hat Linux. Два другие варианта представлены Java-программами

Saxon и

Xalan .

Сгенерировать высококачественный XHTML-документ из любого DocBook-текста сравнительно просто. Большую роль в данном случае играет тот факт, что XHTML является просто другим XML DTD-определением. Преобразование в HTML-документ осуществляется путем применения довольно простой таблицы стилей. Данный способ позволяет также просто генерировать RTF-документы, а из XHTML или RTF легко создать простой ASCII-текст.

Сложный случай представляет собой печать. Создание высококачественного распечатываемого вывода, что на практике означает преобразование в формат Adobe PDF (Portable Document Format), затруднено. Правильное решение данной задачи требует алгоритмического воспроизведения способа мышления наборщика (человека) при переходе от содержания к уровню представления.

Таким образом, прежде всего, с помощью таблицы стилей структурированная разметка DocBook преобразовывается в другой диалект XML — FO (Formatting Objects — объекты форматирования). FO-разметка является ярко выраженной разметкой уровня представления. Ее можно представить как некоторый функциональный XML-эквивалент

troff . Данную разметку необходимо преобразовывать в PostScript для упаковки в PDF.

В инструментальной связке, поставляемой с Red Hat Linux, данная задача решается с помощью TEX-макропакета, который называется PassiveTeX. Данный макрос преобразовывает объекты форматирования, генерируемые xsltproc, в язык TEX Дональда Кнутта. Затем вывод TEX, называемый форматом DVI (DeVice Independent), преобразовывается в PDF.

Сложная цепь XML, TEX-макрос, DVI, PDF — неуклюжая конструкция. Она "гремит, хрипит и имеет уродливые наросты". Шрифты являются значительной проблемой, поскольку XML, TEX и PDF имеют очень разные модели. Кроме того, серьезные трудности связаны с поддержкой интернационализации и локализации. Единственной привлекательной особенностью данной конструкции является то, что она работает.

Изящным способом разрешения описываемой проблемы будет использование FOP, непосредственного преобразования FO в PostScript, разрабатываемого проектом Apache. С помощью FOP проблема интернационализации даже если и не будет решена полностью, то,

по крайней мере, будет определена. XML-инструменты поддерживают Unicode на всем пути к FOP. Преобразование глифов Unicode в PostScript-шрифт является также исключительно проблемой FOP. Единственным недостатком данного подхода является то, что он пока не работает. В середине 2003 года FOP находился в незаконченной альфа-стадии. FOP-преобразование можно использовать, однако оно содержит множество "необработанных углов" и характеризуется недостатком функций.

На рис. 18.3 иллюстрируется схема инструментальной связки FOP.

Рис. 18.3. Будущая инструментальная связка XML-DocBook с использованием FOP

У FOP есть конкурент. Другой проект, который называется

xsl-fo-proc , предназначен для решения тех же задач, что и FOP, но написан на C++ (и, следовательно, работает быстрее, чем Java, и не зависит от Java-окружения). В середине 2003 года проект

xsl-fo-proc находился в незавершенной альфа-стадии и продвинулся не дальше, чем FOP.

18.5.4. Средства преобразования

Вторая крупнейшая проблема, связанная с DocBook, состоит в необходимости преобразования старой разметки уровня представления в разметку DocBook. Человек обычно может автоматически преобразовать представление документа в логическую структуру, поскольку (например) из контекста можно понять, когда курсив означает "акцентирование мысли", а когда он означает что-нибудь другое, например, "иноязычный оборот".

Так или иначе, при конвертировании документов в DocBook данные различия должны быть указаны явно. Иногда они присутствуют в старой разметке. Часто их нет, и отсутствующая структурная информация должна быть либо выведена с помощью развитой эвристики, либо задана человеком.

Ниже приводится сводное описание инструментов преобразования из других форматов. Ни один из описанных ниже инструментов не выполняет данную работу идеально; после преобразования требуется проверка, а возможно, и некоторое редактирование со стороны человека.

GNU Texinfo

Фонд свободного программного обеспечения намеревается поддерживать DocBook как общий формат для обмена информацией. Texinfo имеет достаточную структуру для того, чтобы сделать возможным довольно точное автоматическое преобразование (после преобразования требуются немногочисленные исправления пользователем), а в версиях 4.x

makeinfo представлен ключ --docbook, который генерирует DocBook-документ. Более подробная информация представлена на странице проекта

makeinfo <http://www.gnu.org/directory/texinfo.html>.

POD

Модуль POD::DocBook, <http://www.cpan.org/modules/by-module/Pod/> преобразовывает разметку POD (Plain Old Documentation) в DocBook. Утверждается, что модуль

преобразовывает каждый POD-тег, кроме курсивного тега L<>. На странице руководства также сказано: "Вложенные списки =over/=back внутри DocBook не поддерживаются", однако, следует отметить, что модуль интенсивно тестируется.

LATEX

Проект, который называется TeX4ht, <http://www.lrz-muenchen.de/services/software/sonstiges/tex4ht/mn.html>, способен, по утверждению автора PassiveTEX, генерировать DocBook из LATEX.

man-страницы и другая

troff -разметка

Как правило, считается, что преобразование таких документов представляет крупнейшую и труднейшую проблему. Действительно, базовая разметка

troff(1) находится на слишком низком уровне представления для инструментов автоматического преобразования. Однако ситуация значительно прояснятся, если рассмотреть преобразование из исходных текстов документов, написанных в таких макропакетах, как

man(7) . Они имеют достаточно структурных элементов, для того чтобы улучшить автоматическое преобразование.

Я сам написал инструмент для troff-DocBook преобразования, поскольку не находил другого, который выполнял бы такую работу с приемлемым качеством. Программа называется doclifter, <http://www.catb.org/~esr/doclifter/>. Она транслирует либо в SGML, либо в XML DocBook из макросов

man(7), mdoc(7), ms(7) или

me(7) . Подробности описаны в документации.

18.5.5. Инструменты редактирования

К середине 2003 года отсутствовал один компонент — хорошая программа с открытым исходным кодом для редактирования структуры SGML/XML-документов.

LyX (<http://www.lyx.org/>) представляет собой текстовый процессор с графическим пользовательским интерфейсом, в котором для печати используется LATEX, а также поддерживается структурное редактирование LATEX-разметки. Существует LATEX-пакет, который генерирует DocBook, а также how-to-документ <http://bgu.chez.tiscali.fr/doc/db4lyx/>, описывающий методику написания SGML и XML в LyX GUI.

GNU TeXMacs <http://www.math.u-psud.fr/~anh/TeXmacs/TeXmacs.html> — проект, направленный на создание хорошего редактора для технических и математических материалов, включая отображаемые формулы. Версия 1.0 была выпущена в апреле 2002 года. Разработчики планируют в будущем включить поддержку XML, но в настоящее время ее пока нет.

Большинство пользователей до сих пор редактируют DocBook-теги вручную, используя

vi или

emacs .

18.5.6. Связанные стандарты и практические приемы

Для редактирования и форматирования DocBook-разметки инструменты объединяются. Однако сам по себе формат DocBook является средством, а не целью. Кроме DocBook необходимы другие стандарты для достижения поставленной цели — базы данных документации с возможностью поиска. Существует два больших вопроса: каталогизация документов и метаданные.

Непосредственно на достижение данной цели направлен проект ScrollKeeper <http://scrollkeeper.sourceforge.net/>. Данный проект предоставляет набор сценариев, которые могут использоваться в правилах инсталляции и деинсталляции пакетов для регистрации и удаления их документации.

Программа ScrollKeeper использует открытый формат метаданных (Open Metadata Format) <http://www.ibiblio.org/osrt/omf/>. Данный формат является стандартом для индексирования документации по программам с открытым исходным кодом, аналогичный библиотечной карточно-каталоговой системе. Идея заключается в поддержке развитых средств поиска, в которых используются карточно-каталоговые метаданные, а также исходные тексты документации.

18.5.7. SGML

В предыдущих разделах умышленно опускалась большая часть истории развития формата DocBook. Язык XML имеет "старшего брата", SGML (Standard Generalized Markup Language — стандартный обобщенный язык разметки).

До середины 2002 года любая дискуссия о DocBook была бы неполной без длинного экскурса в SGML, объяснения различий между SGML и XML, а также подробного описания инструментальной связки SGML DocBook. Теперь все значительно проще. Инструментальная связка XML DocBook доступна в виде открытого исходного кода, работает так же, как работала связка SGML, и является более простой в использовании.

18.5.8. Справочные ресурсы по XML-DocBook

Одним из факторов, который затрудняет изучение DocBook, является то, что сайты, связанные с данной темой, часто перегружают новичков длинными списками W3C-стандартов, массивными упражнениями по SGML-идеологии и абстрактной терминологией. Хорошее общее введение представляет собой книга

"XML in a Nutshell" [32].

Книга Нормана Уолша (Norman Walsh)

"DocBook: The Definitive Guide" доступна в печатном виде <http://www.oreilly.com/catalog/docbook/> и в Web — <http://www.docbook.org/tdg/en/html/docbook.html>. Книга представляет собой действительно полный справочник, но в качестве вводного курса или учебного пособия неудобна. Вместо нее рекомендуются материалы, перечисленные ниже.

Написание документов с помощью DocBook (Writing Documents Using DocBook) <http://xml.Web.cern.ch/XML/goossens/dbatcern/> — превосходное учебное пособие.

Существует также одинаково полезный список часто задаваемых вопросов (DocBook FAQ), <http://www.dpawson.co.uk/docbook/> с большим количеством материалов по форматированию HTML-вывода. Кроме того, функционирует DocBook-форум (DocBookwiki) <http://docbook.org/wiki/moin.cgi>.

Наконец, в документе "The XML Cover Pages" <http://xml.coverpages.org/> представлены описания XML-стандартов.

18.6. Лучшие практические приемы написания Unix-документации

Рекомендацию, данную в начале главы, можно рассмотреть с противоположной точки зрения. Создавая документацию для пользователей внутри Unix-культуры,

не следует "оглуплять" ее . Автор документации, написанной для идиотов, сам будет "списан со счетов" как идиот. "Оглупление" документации резко отличается от создания доступной документации. В первом случае автор документации ленив и опускает важные сведения, тогда как во втором требуется глубокое осмысление и безжалостное редактирование.

Не следует, однако, полагать, что объем является ошибкой с точки зрения качества. И что особенно важно —

никогда не следует опускать функциональные детали, опасаясь, что они могут запутать читателя. Не стоит также опускать предупреждения об имеющихся проблемах, чтобы продукт не выглядел плохо. Именно

неожиданные проблемы, а не проблемы, которые разработчик признает открыто, будут стоить ему доверия и пользователей.

Пытайтесь найти золотую середину при подаче информации. Слишком малая насыщенность так же неудачна, как и слишком большая. Расчетливо используйте снимки с экранов; часто они несут мало информации, выходящей за рамки восприятия пользовательского интерфейса, и никогда полностью не заменят хорошего текстового описания.

Если разрабатываемый проект имеет значительные размеры, следует, вероятно, поставлять документацию в трех видах: man-страницы в качестве справочного материала, учебный материал, а также список часто задаваемых вопросов (Frequently Asked Questions — FAQ). Также следует создать Web-сайт, который будет служить центральной точкой распространения продукта (основные принципы взаимодействия с другими разработчиками изложены в главе 19).

Большие man-страницы несколько раздражают пользователей; навигация в них может быть затруднена. Если man-страницы получаются большими, следует рассмотреть возможность

написания справочного руководства, а в man-страницах предоставить краткое обобщение и указатели на справочный материал, а также подробности вызова программы (или программ).

В исходный код следует включать стандартные файлы метаинформации, такие как README, которые описаны в разделе главы 19, касающемся практических приемов выпуска программ с открытым исходным кодом. Даже если предполагается, что разрабатываемый код будет закрытым, эти файлы подчиняются соглашениям Unix и будущие кураторы, приходящие из Unix-среды, быстро вникнут в суть.

man-страницы должны быть авторитетными справочниками в традиционном Unix-стиле для обычной Unix-аудитории. Учебные пособия должны быть документацией в развернутой форме для нетехнических пользователей. FAQ-списки должны быть развивающимися ресурсами, которые растут по мере того, как группа поддержки программы выясняет часто задаваемые вопросы и ответы на них.

Существуют более специфические привычки, которые необходимо развить современному разработчику.

1. Поддержка главных документов в XML-DocBook. Даже тап-страницы могут быть документами DocBook RefEntry. Существует очень хорошее методическое руководство (HOWTO) <http://www.linux.doc.org/HOWTO/mini/Man-Page.html> по созданию man-страниц, в котором описана общая организация, ожидаемая пользователями.

2. Поставка документации с главными XML-файлами. На случай, если системы пользователей не имеют

xmlto(1) , поставляйте исходные тексты

troff , которые можно получить, выполнив команду xmlto man над главными документами. Процедура установки дистрибутива должна устанавливать их обычным способом, однако следует направить пользователей к XML-файлам, если они хотят создавать или редактировать документацию.

3. Создание ScrollKeeper-совместимого инсталляционного пакета проекта.

4. Создание XHTML-документов из главных документов (с помощью команды xmlto xhtml) и предоставление к ним доступа с Web-страницы проекта.

Независимо от того используется ли в качестве главного формата XML-DocBook, необходимо найти способ конвертирования документации проекта в HTML. Независимо от того, как поставляется разрабатываемое программное обеспечение — как открытый исходный код или как частная программа, — растет вероятность того, что пользователи найдут ее посредством Web. Непосредственный эффект от размещения документации в Web-среде заключается в том, что это упрощает ее чтение и изучение для потенциальных пользователей и клиентов, знающих о существовании программы. Кроме того, возникает косвенный эффект — повышается вероятность появления программы в результатах Web-поиска.

19

Открытый исходный код: программирование в новом Unix-сообществе

Программы как секс — лучше, когда они бесплатны. —Линус Торвальдс

Глава 2 заканчивалась формулировкой самого значительного закона в истории Unix. Операционная система Unix "расцветала", когда ее практика наиболее близко приближалась к открытому исходному коду, и приходила в упадок, когда этого не было. Затем в главе 16 утверждалось, что инструменты разработки с открытым исходным кодом часто характеризуются высоким качеством исполнения. Данная глава начинается с общего объяснения того, как и почему действует разработка открытого исходного кода. Большая часть ее поведения является просто усилением общепринятых практических приемов в традиции Unix.

Затем дискуссия выходит из области абстракции, и описываются некоторые из наиболее важных народных традиций, которые Unix заимствовала из сообщества открытого исходного кода, особенно развившиеся в сообществе основные правила относительно того, как должен выглядеть хороший код. Многие из данных традиций могут быть также успешно заимствованы разработчиками в других современных операционных системах.

Данные традиции описываются в предположении, что читатели разрабатывают открытый исходный код. Большинство традиций представляют собой хорошие идеи даже при написании частного программного обеспечения. Предположение об открытом исходном коде является исторически целесообразным, поскольку многие из описываемых традиций через вездесущие инструменты с открытым исходным кодом, такие как

patch(1), Emacs и GCC, "уходят корнями" в частные лаборатории разработки Unix.

19.1. Unix и открытый исходный код

В разработке открытого исходного кода используется тот факт, что выяснение и исправление ошибок, в отличие от, например, реализации определенного алгоритма, является задачей, которая допускает ее разделение на несколько параллельных подзадач. Исследование совокупности возможностей, связанных с конструкцией прототипа, также может осуществляться параллельно. При наличии верного технологического и социального аппарата весьма крупные коллективы разработчиков, которые тесно связаны сетью, способны выполнять поразительно хорошую работу.

Поразительно хорошей она является для тех людей, которые развили в себе ментальную привычку рассматривать секретность процесса и частный контроль как непременное условие. Со времени выхода книги

"The Mythical Man-Month" [8] и до возникновения Linux ортодоксальными в программной инженерии были все мелкие, четко управляемые коллективы внутри тяжеловесных организаций, таких как корпорации и правительство. Чаще всего это были строго управляемые

крупные коллективы.

Раннее Unix-сообщество до лишения прав AT&T было образцовым примером открытого исходного кода в действии. Несмотря на то, что код операционной системы Unix до лишения прав AT&T был технически и юридически частным, он рассматривался как общая часть внутри сообщества пользователей и разработчиков. Добровольные усилия координировались людьми, наиболее сильно заинтересованными в решении проблем. Такая возможность выбора привела к возникновению многих полезных последствий. Действительно, методика

разработки открытого исходного кода развивалась как неосознанная народная практика в Unix-сообществе более чем четверть века, за много лет до того, как она была проанализирована и получила свое название в конце 1990-х годов (см.

"The Cathedral and the Bazaar" [67] и

"Understanding Open Source Software Development" [18]).

Оглядываясь в прошлое, поразительно отмечать, как все мы были склонны забывать последствия нашего поведения. Некоторые подошли весьма близко к пониманию данного феномена. Например, Ричард Габриель (Richard Gabriel) в своей статье "Worse Is Better" [25] в 1990 году, однако прообразы можно найти в книге Брукса [8] (1975 год) и еще более ранних размышлениях Виссотски (Vyssotsky) и Корбати (Corbaty) о конструкции Multics (1965 год). Мне не удавалось это сделать в течение более чем 20 лет наблюдения за разработкой программного обеспечения, пока меня в середине 1990-х годов "не разбудила" Linux. Этот опыт должен заставить любого вдумчивого и сдержанного человека заинтересоваться тем, какие другие важные унифицирующие концепции в нашем поведении до сих пор остаются неявными и проходят незамеченными нашим коллективным разумом, скрытые не из-за своей сложности, а из-за крайней простоты.

Перечислим весьма простые правила разработки открытого исходного кода.

1.

Пусть код будет открытым. Никаких секретов. Сделайте код и процесс его создания достоянием общественности. Поощряйте экспертную оценку третьей стороны. Убедитесь, что другие могут свободно модифицировать и распространять код. Выращивайте как можно большее сообщество разработчиков-соавторов.

2.

Выпускайте первую версию быстро, выпускайте последующие версии часто. Быстрый темп выхода версий означает быструю и эффективную обратную связь. Когда каждая следующая версия является небольшой, изменение курса в ответ на отзывы реальных пользователей будет проще.

Просто убедитесь, что первая версия программы компилируется, работает и демонстрирует перспективы. Обычно первоначальная версия программы с открытым исходным кодом демонстрирует перспективы, выполняя, по крайней мере, некоторую часть своей окончательной работы, достаточную для иллюстрации того, что разработчик может действительно продолжать работу над проектом. Например, первоначальная версия текстового процессора может поддерживать ввод текста и его отображение на экране.

Первая версия, которую невозможно скомпилировать или использовать, может погубить проект (что, как известно, почти произошло с браузером Mozilla). Некомпилирующиеся версии наводят на мысль, что разработчики проекта не способны завершить его. Кроме того, неработающие программы усложняют сотрудничество для других разработчиков, поскольку сложно определить, улучшили ли программу какие-либо внесенные ими изменения.

3. Поощряйте сотрудничество похвалой. Если нет возможности материально вознаградить вносящих свой вклад разработчиков, поощряйте их морально. Даже если возможно наградить человека материально, помните, что люди часто для собственной репутации работают интенсивнее, чем работали бы за деньги.

Следствием второго правила является то, что выпуск отдельных версий не должен быть важным событием, предполагающим большую подготовку и многообещающие перспективы.

Важно беспощадно рационализировать процесс выхода новых версий, с тем чтобы свежие версии

можно было выпускать безболезненно. Схема, при которой во время подготовки новой версии необходимо прекратить всю остальную работу, крайне неудачна. (Особенно в случае использования системы CVS или подобной ей, подготовительные версии должны "ответвляться" от главной линии разработки, для того чтобы не блокировать основной процесс.) Вывод: не следует трактовать выход версии как большое, особое событие; сделайте его частью установившегося процесса.Генри Спенсер.

Необходимо помнить, что причиной для частого выхода новых версий является необходимость сокращения и ускорения цикла обратной связи, который соединяет пользовательскую аудиторию с разработчиками. Следовательно, не следует думать о следующей версии, как о безупречном драгоценном камне, который нельзя показывать до тех пор, пока он не станет идеальным. Не следует создавать длинных перечней пожеланий. Делайте прогресс постепенным, признавайте текущие ошибки и объявляйте о них, и будьте уверены, что совершенство придет со временем. Согласитесь с тем, что придется выпустить десятки мелких версий, и не огорчайтесь, когда номера версий растут.

В разработке открытого исходного кода задействованы большие коллективы программистов, рассеянных по всему миру и общающихся главным образом с помощью электронной почты и Web-страниц. Как правило, большинство соавторов в любом проекте являются добровольцами, вносящими свой вклад ради повышения полезности программы для них самих. Управляет проектом главный разработчик или центральная группа. Другие соавторы могут время от времени подключаться к проекту и выбывать из него. Для того чтобы поощрять соавторов-любителей, важно не допустить появления социальных барьеров между ними и основным коллективом. Необходимо минимизировать привилегированный статус основного коллектива и упорно работать над тем, чтобы сохранить границы незаметными.

Проекты с открытым исходным кодом следуют традиционному для Unix-сообщества совету автоматизировать работу при любой возможности. Разработчики используют утилиту

patch(1) для распространения последовательных изменений. Во многих проектах (и во всех крупных) имеются доступные по сети репозитории кода, использующие системы контроля версий, подобные CVS (эта тема подробно рассматривается в главе 15). Использование автоматизированных систем отслеживания ошибок и исправлений также является широко распространенной практикой.

В 1997 году за пределами хакерской культуры почти никто не понимал, что таким способом можно разрабатывать крупные проекты, позволяя единицам добиваться высококачественных результатов. Проекты, подобные Linux, Apache и Mozilla, одновременно добились успеха и высокой общественной заинтересованности.

Отказ от привычки к секретности в пользу прозрачности процесса и экспертной оценки, был ключевым этапом, который превратил "алхимию" в "химию". Также начали появляться признаки того, что разработка открытого исходного кода может сигнализировать о долгожданном формировании разработки программного обеспечения как дисциплины.

19.2. Лучшие практические приемы при взаимодействии с разработчиками открытого исходного кода

Основной составляющей лучшей практики в сообществе открытого исходного кода является

естественная адаптация к распределенной разработке. В оставшейся части данной главы представлены сведения о поведении, при котором поддерживается хорошая связь с другими разработчиками. Там, где соглашения Unix являются условными (такие как стандартные имена файлов, передающие метаинформацию об исходном дистрибутиве), часто прослеживается их происхождение либо из Usenet (начало 1980-х годов), либо из соглашений и стандартов проекта GNU.

19.2.1. Хорошая практика обмена исправлениями

Большинство людей вовлекаются в создание программного обеспечения с открытым исходным кодом, создавая заплаты для чужих программ, прежде чем создают собственные проекты. Предположим, некто создал набор исправлений исходного кода для основной части чужой программы. Если встать на место разработчика программы, то каким образом можно решить, включать ли данную заплату или нет?

Очень сложно судить о качестве кода, поэтому разработчики склонны оценивать исправления по качеству их подачи. Они анализируют ход мыслей в стиле подачи исправлений и манере общения лица, присылающего заплату.

За много лет работы с заплатами, полученными от сотен незнакомых людей, автор данной книги редко встречал какое-либо исправление, которое было представлено продуманно, с уважением моего времени, но технически было нефункциональным. С другой стороны, опыт свидетельствует, что исправления, которые выглядят неаккуратными или небрежно и поспешно упакованными, скорее всего,

являются неработоспособными.

Ниже приводятся некоторые рекомендации для того, чтобы созданная вами заплата была принята.

19.2.1.1. Отправляйте заплаты, а не целые архивы или файлы

Если изменения включают в себя новый файл, который отсутствует в коде, то, естественно, приходится отправлять данный файл целиком. Однако если изменяются только уже существующие файлы, отправлять их полностью не следует. Вместо этого следует отправить diff-файл, а именно вывод команды

diff(1) , выполненной для сравнения основной распространяемой версии с модифицированной.

Команда

diff(1) и обратная ей команда

patch(1) являются основными инструментами разработки открытого исходного кода. Diff-файлы лучше, чем целые файлы, поскольку разработчик, которому отсылается заплата, мог изменить основной код с момента получения копии создателем заплаты. Отправляя разработчику diff-файл, создатель заплаты предотвращает дополнительную работу по разделению изменений, которую придется выполнить разработчику, и демонстрирует таким

образом, что он ценит его время.

19.2.1.2. Отправляйте исправления к текущей версии кода

Не стоит отправлять куратору заплаты к коду, который существовал за несколько версий до этого, и ожидать, что он сделает всю работу по определению того, какие исправления дублируются с изменениями, уже внесенными им самим, а какие действительно являются новыми.

Ответственность за отслеживание состояния исходного кода и отправку куратору минимальных заплат, которые выражают то, что их создатель хочет изменить в основном коде главной линии разработки, лежит на самом

создателе заплаты. Это означает отправку исправлений к текущей версии.

19.2.1.3. Не следует включать заплаты для генерируемых файлов

Прежде чем отправить заплату, необходимо просмотреть ее и удалить все диапазоны кода, предназначенные для файлов, которые должны генерироваться автоматически после применения куратором исправлений и перекомпиляции программы. Классическим примером данной ошибки являются C-файлы, сгенерированные инструментами

Bison или

Flex .

В настоящее время наиболее распространенной разновидностью данной ошибки является отправка diff-файлов с большим диапазоном, который не содержит ничего, кроме отличий между сценариями configure создателя заплаты и куратора. Такой файл создается с помощью autoconf.

Такой подход является необдуманным и означает, что получатель будет вынужден отделять действительное содержание заплаты от большого количества шума. Данная ошибка незначительна, она не настолько важна, как некоторые из рассматриваемых далее, однако она не делает чести создателю заплаты.

19.2.1.4. Не отправляйте заплат, которые только убирают $-идентификаторы систем RCS или SCCS

Некоторые разработчики вносят специальные метки в свои исходные файлы, распространяемые с помощью системы контроля версий, когда возвращают отредактированные файлы, например, конструкции Id, используемые в системах RCS и CVS.

Если используется локальная система контроля версий, то изменения могут модифицировать данные метки. Это не представляет реальной опасности, поскольку когда получатель снова

зарегистрирует исправленный код после применения заплаты, метки будут переназначены снова в соответствии с состоянием системы контроля версий

куратора . Однако такие диапазоны исправлений являются излишними и отвлекают внимание. Лучше их не отправлять.

Данная ошибка также является незначительной, и вас простят, если более важные изменения сделаны верно, однако ее следует в любом случае избегать.

19.2.1.5. Используйте вместо формата по умолчанию (-e) форматы -c или -u

Принятый по умолчанию в

diff(1) формат -e является крайне ненадежным. Он не включает в себя контекст, поэтому утилита patch не может справиться со своей задачей, если какие-либо строки были вставлены в код главной линии проекта или удалены с момента получения создателем заплаты своей копии кода.

Получение diff-файлов, созданных с ключом -e, раздражает и наводит на мысль, что отправитель либо крайне неопытен, либо неаккуратен, либо неграмотен. Большинство подобных заплат удаляется без размышлений.

19.2.1.6. Сопровождайте заплаты документацией

Данный момент очень важен. Если заплата вносит заметные пользователям дополнения или изменяет функции программы, то

необходимо включить данные изменения в соответствующие man-страницы и другие файлы документации к заплате. Не следует полагать, что получатель будет счастлив документировать чужой код или использовать недокументированные функции в своем коде.

Документирование изменений хорошо демонстрирует некоторые полезные качества. Во-первых, это вежливо по отношению к человеку, которого вы пытаетесь убедить. Во-вторых, это демонстрирует ваше понимание того, что смысл внесенного изменения достаточно важен для того, чтобы объяснять его тем, кто не видит код. В-третьих, это демонстрирует заботу о людях, которые, в конце концов, будут использовать данную программу.

Хорошая документация обычно является наиболее заметным признаком, который позволяет отличить солидный вклад от быстрой и неаккуратной работы. Если созданию документации уделить необходимое время и внимание, то вскоре обнаружится, что пройдено 85% пути к принятию заплаты большинством разработчиков.

19.2.1.7. Сопровождайте заплату пояснениями

Заплата должна включать в себя пояснительную записку, объясняющую необходимость или

практическую пользу заплаты с точки зрения ее создателя. Данное пояснение адресовано не пользователям программного обеспечения, а его куратору, принимающему заплату.

Записка может быть краткой, фактически некоторые из наиболее эффективных пояснительных записок, которые встречались автору, содержали в себе только одну мысль: "См. обновления документации в данной заплате". Однако записка должна демонстрировать правильное отношение к работе.

Правильное отношение полезно, оно демонстрирует уважение к времени куратора; подобные записки весьма конфиденциальны, но непритязательны. Правильно будет показать понимание исправляемого кода, а также то, что создатель заплаты способен разделить проблемы куратора. Также хорошо закончить упоминанием о любых осознаваемых рисках, связанных с применением заплаты. Ниже приводится несколько примеров пояснительных комментариев, которые отправляют опытные разработчики.

"Я обнаружил в коде две проблемы, X и Y. Проблему X я решил, а проблему Y даже не пытался рассматривать, поскольку не думаю, что понимаю ту часть кода, которая, как я полагаю, с ней связана."

"Исправил ошибки по дампу, которые могут возникать, когда какой-либо foo-ввод является слишком длинным. В ходе решения проблемы я искал подобные переполнения в других местах. Возможная причина переполнения находится в файле blarg.c, где-то около строки 666. Вы уверены, что отправитель не может сгенерировать более 80 символов в течение одной передачи?"

"Вы рассматривали использование Foonly-алгоритма для решения данной проблемы? Здесь есть пример хорошей реализации — <http://www.example.com/~jsmith/foonly.html>."

"Данная заплата решает первоочередную проблему, но я понимаю, что она неприятно усложняет распределение памяти. У меня заплата работает, однако перед распространением ее, вероятно, следует протестировать под большой нагрузкой."

"Возможно, это — украшательство, но я в любом случае его отправляю. Может быть, вам известен более четкий способ реализации данной функции".

19.2.1.8. Включайте в код полезные комментарии

Куратор хочет иметь полную уверенность в том, что понимает вносимые заплатой изменения, прежде чем объединить их с основным кодом. Данное правило не является неизменным. Если создатель заплаты имеет стаж хорошей работы с куратором, то куратор может только бегло просмотреть изменения перед их почти автоматическим применением. Но все, что можно сделать, чтобы помочь ему понять код и уменьшить неуверенность, увеличивает вероятность того, что заплата будет принята.

Хорошие комментарии в коде помогают куратору понять его, а плохие мешают.

Ниже приводится пример плохого комментария.

/* норман ньюби исправил этот код 13 августа 2001 года */

Данный комментарий не несет в себе никакой информации — просто помарка в центре кода куратора. Если куратор примет данную заплату (что вряд ли), то почти гарантированно удалит данный комментарий. Если разработчик хочет добиться доверия, то ему следует включить

диапазон исправлений для таких файлов проекта, как NEWS или HISTORY. В таком случае принятие заплаты куратором более вероятно. Ниже приведен пример хорошего комментария.

```
/*
* Необходимо защитить этот переход, так чтобы в crunch_data()
* никогда не передавался NULL-указатель.
 &lt;norman_newbie@foosite.com&gt;
*/
```

Данный комментарий показывает, что вы понимаете не только код куратора, но и ту информацию, которая ему необходима для того, чтобы быть уверенным в ваших изменениях. Такой вид комментария

предоставляет ему такую возможность.

19.2.1.9. Не огорчайтесь, если заплата отклонена

Существует множество причин отклонения заплат, которые никак не связаны с их создателями. Нельзя забывать, что большинство кураторов крайне ограничены во времени и должны быть консервативными в вопросе приема заплат, чтобы не нарушить код проекта. Иногда повторная подача с уточнениями помогает, иногда нет. Такова жизнь.

19.2.2. Хорошая практика наименования проектов и архивов

По мере роста нагрузки на кураторов архивов, подобных ibiblio, SourceForge и CPAN, возрастает тенденция к обработке заявок частично или полностью программным путем (вместо полной их проверки вручную).

В связи с этим возрастает важность подчинения имен проектов и архивов обычным шаблонам, которые могут анализироваться и распознаваться программами.

19.2.2.1. Используйте GNU-стиль названий с именной частью и номерами (основной.второстепенный.заплата)

Будет хорошо, если архивные файлы проекта будут иметь GNU-подобные названия — именной префикс, состоящий из строчных алфавитно-цифровых символов, дефис, номер версии, расширение и другие суффиксы.

Хорошая общая форма названия содержит следующие части в указанном порядке.

1. Префикс проекта.

2. Дефис.

3. Номер версии.

4. Точка.

5. "src" или "bin" (не обязательно).

6. Точка или дефис (точка предпочтительнее).

7. Тип бинарных файлов и параметры (не обязательно).

8. Расширения архивирования и компрессии.

Именная часть в данном стиле может содержать дефис или подчеркивание для разделения составляющих слов; в настоящее время использование дефисов предпочтительнее. Хорошей практикой является группировка связанных проектов путем включения в именные части общего префикса, ограниченного дефисом.

Предположим, существует проект, который называется "foobar", основной номер версии (major version) 1, второстепенный номер версии (minor version) или выпуска 2, уровень исправлений (patchlevel) 3. Ниже представлены примеры названий для проекта, содержащего только один архив (предположительно исходный код).

foobar-1.2.3.tar.gz

Архив исходного кода.

foobar.lsm

LSM-файл (для публикации в ibiblio).

Не следует использовать имена, подобные приведенным ниже.

foobar123.tar.gz

Многие программы распознают такое имя как архив для проекта, который называется "foobar123" без номера версии.

foobar1.2.3.tar.gz

Многие программы распознают такое имя как архив для проекта, который называется "foobar1" версии 2.3.

foobar-v1.2.3.tar.gz

Многие программы распознают такое имя как проект, который называется "foobar-v1".

foo_bar-1.2.3.tar.gz

Подчеркивание плохо передается в устной речи, его неудобно вводить и запоминать.

FooBar-1.2.3.tar.gz

Если только разработчик

не хочет выглядеть как знаток маркетинга. Такая форма также трудна для устного выражения, ввода и запоминания.

Если необходимо дифференцировать архивы исходного кода и бинарных файлов или различные виды бинарных файлов, либо выразить в имени файла некоторые параметры компиляции, то данную информацию следует интерпретировать как расширение файла, следующее

после номера версии. Примеры приведены ниже.

foobar-1.2.3.src.tar.gz

Исходный код.

foobar-1.2.3.bin.tar.gz

Бинарные файлы без указания типа.

foobar-1.2.3.bin.i386.tar.gz

Бинарные файлы для архитектуры i386.

foobar-1.2.3.bin.i386.static.tar.gz

Статически связанные бинарные файлы для архитектуры i386.

foobar-1.2.3.bin.SPARC.tar.gz

Бинарный код для архитектуры SPARC.

Не используйте такие имена, как "foobar-i386-1.2.3.tar.gz", поскольку программы испытывают затруднения с выделением инфиксов типа (например, "-i386") из именной части.

Соглашение о различиях основной и второстепенной нумерации версий весьма простое: уровень заплат увеличивается на единицу для исправлений или внесения незначительных функций, второстепенный номер версии — для совместимых новых функций, а основной номер версии увеличивается, когда сделаны несовместимые изменения.

19.2.2.2. По возможности необходимо придерживаться локальных соглашений

В некоторых проектах и сообществах имеются четкие соглашения для имен и номеров версий, которые не обязательно согласуются с приведенными выше рекомендациями. Например, модули Apache, как правило, получают такие имена, как "modfoo", и имеют номер собственной версии и номер версии Apache, с которой они работают. Подобным образом, модули Perl имеют номера версий, которые можно интерпретировать как числа с плавающей точкой (например, можно встретить версию 1.303, а не 1.3.3), а дистрибутивы обычно получают такие имена, как "Foo-Bar-1.303.tar.gz", для версии 1.303 модуля Foo::Bar. (Сам проект Perl, с другой стороны, перешел на использование описанных здесь соглашений в конце 1999 года.)

Ищите и следуйте соглашениям определенных сообществ и разработчиков; для общего использования придерживайтесь описанных выше принципов.

19.2.2.3. Упорно ищите уникальный префикс имени, который легко вводить

Основной префикс должен быть общим для всех файлов проекта, а также должен легко читаться, вводиться и запоминаться, поэтому не следует использовать символ подчеркивания. Также не следует без крайне важной причины использовать прописную букву или две прописные буквы — это вносит путаницу в естественный порядок зрительного поиска и выглядит так, как будто новичок маркетинга пытается показаться умнее.

Два различных проекта, имеющих одинаковые именные части, сбивают людей с толку. Поэтому, прежде чем выпускать первую версию, необходимо проверить имя на предмет конфликтов. Для такой проверки можно использовать индексный файл проекта ibiblio <http://metalab.unc.edu/pub/Linux> и индекс приложений на сайте Freshmeat <http://www.freshmeat.net>. Хорошим ресурсом для проверки также является сайт проекта SourceForge <http://www.sourceforge.net>.

19.2.3. Хорошая практика разработки

Ниже описаны модели поведения, которые могут отличать успешный проект с большим количеством разработчиков от проекта, остановившегося после того, как он не привлек к себе интереса.

19.2.3.1. Не полагайтесь на частный код

Не полагайтесь на частные языки, библиотеки или другой код, так как это в большинстве случаев рискованно. В сообществе открытого исходного кода такой подход считается неприкрытой грубостью. Разработчики открытого исходного кода не доверяют коду, который невозможно просмотреть.

19.2.3.2. Используйте автоинструменты GNU

Определение конфигурационных параметров должно быть выполнено во время компиляции. Значительное преимущество дистрибутивов с открытым исходным кодом заключается в том, что они позволяют адаптировать пакет к обнаруженной среде на этапе компиляции. Это особенно важно, поскольку позволяет пакету работать на таких платформах, которые были недоступны разработчикам пакета, а также позволяет сообществу пользователей данной программы создавать собственные версии. Только крупнейшие коллективы разработчиков могут позволить себе купить все аппаратное обеспечение и нанять достаточное количество работников для поддержки даже ограниченного числа платформ.

Следовательно: для того чтобы решить проблемы переносимости, определения системной конфигурации и адаптации make-файлов, необходимо использовать автоматические инструменты проекта GNU. Сегодня пользователи, устанавливающие программы из исходных кодов, ожидают возможности ввести команды configure; make; make install и получить чистую сборку, и это действительно так. Полезные учебные материалы по работе с данными инструментами приведены на странице <http://seul.org/docs/autotut>.

Зрелыми утилитами являются

autoconf и

autoheader . Программа

automake , как отмечалось ранее, еще до середины 2003 года была нестабильной и содержала много ошибок. Возможно, понадобится поддерживать собственный файл Makefile.in. К счастью,

automake является наименее важной программой в наборе автоинструментов.

Независимо от подхода к конфигурации, не следует задавать пользователю вопросы по конфигурации системы на этапе компиляции. Пользователь, устанавливающий пакет, не знает ответов на эти вопросы, и такой подход обречен с самого начала. Программа должна быть способна определить любые необходимые ей данные во время компиляции или установки.

Однако утилиту

autoconf нельзя рассматривать как одобрение кнопочных конструкций. Если вообще возможно, при программировании необходимо придерживаться стандартов, подобных POSIX, и воздерживаться от опроса системы для получения конфигурационной информации. Рекомендуется сохранять минимальное число ifdefs-директив, а еще лучше — не иметь их вообще.

19.2.3.3. Тестируйте код перед выпуском версии

Хороший тестовый комплект позволяет коллективу легко выполнять возвратные тесты перед выпуском новых версий. Рекомендуется создавать устойчивую, полезную структуру теста, для того чтобы можно было последовательно добавлять в программу тесты без необходимости обучать разработчиков специфическим сложностям тестового комплекта.

Распространение тестового комплекта позволяет сообществу пользователей проверить свои версии перед отправкой пожеланий группе разработчиков.

Поощряйте разработчиков к использованию широкого многообразия платформ в качестве настольных и тестовых машин, для того чтобы непрерывная проверка кода на предмет дефектов переносимости стала частью обычной разработки.

Хорошая практика, подкрепляющая уверенность в коде, заключается в поставке кода с тестовым комплектом, который используется разработчиком и который можно запустить с помощью команды make test.

19.2.3.4. Выполняйте контроль ошибок в коде перед выпуском версии

Под "контролем ошибок" (sanity check) здесь подразумевается использование всех доступных инструментов, обладающих приемлемой способностью к обнаружению ошибок, которые человек склонен пропускать. Чем больше таких ошибок обнаружат данные инструменты, тем

меньше пользователям и самому разработчику придется с ними бороться.

При написании программ на C/C++ с использованием GCC рекомендуется выполнять тестовую компиляцию с параметром -Wall и устранять все ошибки перед каждым выходом новой версии. Кроме того, стоит компилировать код всеми доступными компиляторами — разные компиляторы часто обнаруживают различные проблемы. В частности, скомпилируйте программу на машине с действительно 64-битовой архитектурой. Базовые типы данных могут изменяться на 64-битовых машинах, и поэтому в них часто обнаруживаются новые проблемы. Найдите систему Unix-поставщика и запустите утилиту lint для проверки программы.

Используйте инструменты, которые ищут утечки памяти и другие ошибки времени выполнения. Программы Electric Fence и Valgrind — хорошие инструменты, доступные в виде открытого исходного кода.

Для Python-проектов полезным инструментом проверки может оказаться программа PyChecker <http://sourceforge.net/projects/pychecker>. Она часто обнаруживает нетривиальные ошибки.

При написании программ на Perl проверять код следует с помощью ключа -c (и возможно -T, если он применим). Используйте ключ -w и конструкции "use strict". (Дальнейшую информацию можно найти в документации на Perl.)

19.2.3.5. Проверяйте орфографию в документации и README-файлах перед выпуском версии

Проверяйте грамотность документации, README-файлов и сообщений об ошибках в программе. Сырой код, т.е. код, который вызывает появление сообщений об ошибках при компиляции и имеет орфографические ошибки в текстах README-файлов и предупреждений, приводит пользователей к мысли, что проектирование данной программы также случайно и бессистемно.

19.2.3.6. Рекомендованные практические приемы переносимости кода C/C++

При написании программ на C используйте полные ANSI-функции. В частности, используйте прототипы функций, которые помогают выявить несовместимость между модулями. Старые компиляторы в стиле K&R — древняя история.

Не полагайтесь на специфические для некоторых компиляторов функции, такие как GCC-параметр -pipe или вложенные функции. Они впоследствии повлияют на чужие порты в не-Linux и Не-GCC-системе.

Необходимый для обеспечения переносимости код должен быть изолирован в отдельной области и отдельном наборе исходных файлов (например, в подкаталоге os). Компилятор, библиотека и интерфейсы операционной системы с проблемами переносимости должны быть абстрагированы в файлы данного каталога.

Уровень переносимости представляет собой библиотеку (или, возможно, просто набор макросов в заголовочных файлах), которая извлекает только части API-интерфейсов операционных систем, в которых нуждается разрабатываемая программа. Уровни

переносимости облегчают создание версий программы для других платформ. Часто никто из членов коллектива разработчиков не знает целевую платформу (например, существуют буквально сотни различных встроенных операционных систем, и никто незнаком со значительной частью этих платформ). Создание отдельного уровня переносимости предоставляет специалисту, знающему целевую платформу, возможность переносить программу без необходимости понимания чего-либо за пределами уровня переносимости.

Уровни переносимости также упрощают приложения. Программы редко нуждаются в полной функциональности или более сложных системных вызовах, таких как mmap

(2) или

stat(2) , а программисты часто конфигурируют такие сложные интерфейсы неверно. Уровень переносимости с абстрактными интерфейсами (например, функция с именем __file_exists вместо вызова

stat(2)) позволяет импортировать из системы только ограниченную, необходимую функциональность, упрощая код приложения.

Рекомендуется всегда писать уровень переносимости на основе функции, а не на основе платформы. Попытки создания отдельных уровней переносимости для каждой поддерживаемой платформы приводят к многочисленным проблемам обновления и обслуживания. "Платформа" всегда выбирается на основе по крайней мере двух параметров: компилятор и версия библиотеки/операционной системы. В некоторых случаях используются три фактора, как когда Linux-поставщики выбирают библиотеку C независимо от версии операционной системы. В случае с

M -поставщиками,

N -компиляторами и

O -версиями операционных систем, количество платформ быстро превышает пределы досягаемости любых коллективов разработчиков, кроме крупнейших. С другой стороны, при использовании стандартов языков и систем, таких как ANSI и POSIX 1003.1, набор функций является относительно ограниченным.

Выбор уровня переносимости можно осуществлять либо по строкам кода, либо по компилированным файлам. Нет различий при выборе альтернативных строк кода на платформе или одного из нескольких различных файлов. Практическое правило заключается в том, чтобы перенести код переносимости для различных платформ в отдельные файлы, когда реализация значительно отличается (распределение общей памяти в Unix и Windows), и оставлять код переносимости в одном файле, когда различия минимальны (например, в зависимости от использования функции gettimeofday, clock_gettime, ftime или time для определения текущего времени суток).

За пределами уровня переносимости необходимо придерживаться следующей рекомендации.

Директивы #ifdef и #if являются последним средством, обычно признаком неудачного воображения, чрезмерной дифференциации продукта, неоправданной "оптимизации" или накопленного мусора. Их использование в середине кода — бедствие. Образцовый пример — /usr/include/stdio.h из GNU. Дуг Макилрой.

Использование директив #ifdef и #if допустимо (если хорошо контролируется) внутри уровня переносимости. За его пределами необходимо упорно пытаться заключить их в обусловленные #includes, исходя из символов функций.

Никогда не следует вторгаться в пространство имен любой другой части системы, включая имена файлов, возвращаемые коды ошибок и имена функций. Там где пространство имен используется совместно, необходимо документировать используемую программой часть.

Выбирайте стандарт кодирования. Споры вокруг выбора стандарта можно продолжать вечно — независимо от этого очень трудно и дорого обслуживать программное обеспечение, созданное с использованием нескольких стандартов кодирования, поэтому необходимо выбрать некоторый общий стиль. Безжалостно приводите в действие избранный стандарт кодирования, поскольку последовательность и чистота кода имеют наивысший приоритет. Подробности стандарта кодирования второстепенны.

19.2.4. Хорошая практика создания дистрибутивов

Приведенные ниже рекомендации описывают, как должен выглядеть дистрибутив, когда пользователь загружает, восстанавливает и распаковывает его.

19.2.4.1. Убедитесь, что архивы всегда распаковываются в один новый каталог

Наиболее досадной ошибкой неопытных соавторов является сборка архивов, которые распаковывают файлы и каталоги дистрибутива в текущий каталог, что связано с потенциальной возможностью перезаписи имеющихся в нем файлов.

Такой подход не рекомендуется .

Вместо этого следует убедиться, что все архивные файлы имеют общий каталог, который назван именем проекта, для того чтобы они распаковывались в один каталог верхнего уровня непосредственно в текущем каталоге. Традиционно имя каталога должно быть таким же, как именная часть архива. Например, предполагается, что архив с именем foo-0.23.tar.gz распаковывается в подкаталог foo-0.23.

В примере 19.1 демонстрируется решение для make-файла, которое позволяет реализовать указанный принцип в предположении, что каталог дистрибутива называется "foobar", а SRC содержит список файлов дистрибутива. Пример 19.1. make-правила для tar-архива

```
foobar-$(VERS) .tar.gz:
 @ls $(SRC) | sed s:^:foobar-$(VERS)/: &gt;MANIFEST
 @(cd ..; ln -s foobar foobar-$(VERS))
 (cd ..; tar -czvf foobar/foobar-$(VERS).tar.gz `cat foobar /MANIFEST`)
 @(cd ..; rm foobar-$(VERS))
```

19.2.4.2. Включайте в дистрибутив README-файл

В дистрибутив программы следует включать файл README, который является путеводителем по дистрибутиву. Согласно давнему соглашению (созданному самим Деннисом Ритчи до 1980 года, и распространенному в Usenet в начале 1980-х годов), данный файл является первым файлом, который будут читать бесстрашные исследователи, после того как распакуют исходный код.

README-файлы должны быть короткими и легко читаемыми. Они должны быть вводным документом, а не крупным произведением. Рассмотрим пункты, которые рекомендуется включать в README-файлы.

1. Краткое описание проекта.

2. Ссылка на Web-сайт проекта (если он существует).

3. Примечания по среде разработки и потенциальным проблемам переносимости.

4. Схема проекта, описывающая важные файлы и подкаталоги.

5. Инструкции по компиляции/установке или ссылка на файл, содержащий такие инструкции (обычно INSTALL).

6. Список кураторов/благодарностей или ссылка на содержащий его файл (обычно CREDITS).

7. Последние новости проекта или ссылка на содержащий их файл (обычно NEWS).

8. Адреса списков рассылки проекта.

В свое время данный файл обычно назывался READ.ME, однако с таким именем плохо взаимодействуют браузеры, которые слишком склонны интерпретировать файл с суффиксом .ME как нетекстовый и позволяют только загружать его, а не просматривать. Такой подход устарел и уже не рекомендуется.

19.2.4.3. Придерживайтесь стандартной практики именования файлов

Еще до просмотра README-файла бесстрашный исследователь внимательно изучит имена файлов в корневом каталоге распакованного дистрибутива. Имена файлов в нем сами по себе способны нести полезную информацию. Соблюдая определенные стандартные принципы наименования, разработчик может дать исследователю ценную подсказку о том, "куда смотреть дальше".

Ниже приведены некоторые стандартные имена файлов верхнего уровня, а также описания соответствующих файлов. Не каждый дистрибутив нуждается во всех перечисленных файлах.

README

Файл-путеводитель, который прочитывают первым.

INSTALL

Инструкции по конфигурированию, компиляции и установке.

AUTHORS

Перечень участников проекта (GNU-соглашение).

NEWS

Последние новости проекта.

HISTORY

История проекта.

CHANGES

Перечень значительных изменений между различными редакциями.

COPYING

Лицензионные условия проекта (GNU-соглашение).

LICENSE

Лицензионные условия проекта.

FAQ

Текстовый документ с перечнем часто задаваемых вопросов по проекту.

Следует отметить существование общего соглашения, согласно которому имена файлов, набранные в верхнем регистре, содержат метаинформацию о проекте и предназначены для чтения людьми, а не являются компонентами сборки. Данное уточнение файлов README было давно разработано Фондом свободного программного обеспечения.

Наличие файла FAQ может предотвратить множество неприятностей. Когда какой-либо вопрос по проекту встречается часто, его следует поместить в список часто задаваемых вопросов, после чего предложить пользователям прочесть данный файл, прежде чем отправлять свои вопросы или отчеты об ошибках. Хорошо организованный FAQ-файл способен на порядок или больше снизить нагрузку по поддержке, которая лежит на кураторах проекта.

Весьма ценным является наличие файлов HISTORY или ы, содержащих временные метки для каждой версии. Кроме всего прочего, такие файлы могут помочь доказать существование предшествующей разработки, если разработчику когда-либо придется столкнуться с судебным процессом по нарушению патентов (такое еще не случалось, но лучше быть готовым).

19.2.4.4. Проектирование с учетом обновлений

По мере выхода новых версий, программы изменяются. Некоторые изменения обратно не совместимы. Соответственно, следует серьезно продумать проектирование схемы инсталляции, для того чтобы несколько установленных версий кода могли сосуществовать в одной системе. Это особенно важно в отношении библиотек — нельзя рассчитывать на то, что все клиентские программы будут обновляться по мере изменения вашего API-интерфейса.

Проекты Emacs, Python и Qt имеют хорошее соглашение для реализации этого принципа:

каталоги с номерами версий (другой практический прием, который, вероятно, становится установившейся практикой благодаря FSF). Ниже показано, как выглядит установленная библиотека Qt (${ver} — номер версии).

/usr/lib/qt /usr/lib/qt-${ver}

/usr/lib/qt-${ver}/bin # Расположение moc

/usr/lib/qt-${ver}/lib # Расположение .so

/usr/lib/qt-${ver}/include # Расположение заголовочных файлов

При наличии такой организации сосуществование нескольких версий вполне возможно. Клиентские программы должны указывать необходимую версию библиотеки, однако это небольшая цена, которую приходится уплатить, чтобы не иметь сбоев в интерфейсах. Данная практика позволяет избежать печально известного "ада DLL", характерного для Windows.

19.2.4.5. В Linux создавайте RPM-пакеты

Де-факто стандартным форматом для устанавливаемых бинарных пакетов в Linux является формат, используемый диспетчером пакетов Red Hat Linux, RPM (Red Hat Package manager). Он имеется в большинстве популярных дистрибутивов Linux и поддерживается фактически всеми остальными дистрибутивами (кроме Debian и Slackware; а в Debian можно устанавливать программы из RPM-пакетов). Следовательно, хорошей идеей для сайта проекта будет предоставление устанавливаемых RPM-пакетов наряду с архивами исходных кодов.

Также хорошей идеей будет включение в архив с исходным кодом файл RPM-спецификации, из которого правило в makefile позволяет создавать RPM-пакеты. Файл спецификации должен иметь расширение .spec; по данному расширению команда rpm -t обнаруживает его в архиве.

Для особых целей рекомендуется генерировать файл спецификации с shell-сценарием, который автоматически вставляет корректный номер версии, анализируя файлы makefile или version.h проекта.

Следует отметить, что в случае поставки исходных RPM-пакетов необходимо использовать тег BuildRoot, для того чтобы программа компилировалась в каталоге /tmp или /var/tmp. Если этого не сделать, то во время выполнения инсталляционной части компиляции файлы будут установлены в действительно конечный каталог. Это произойдет, даже если обнаружатся файловые коллизии, а также если пользователь вообще не хочет инсталлировать пакет. После завершения процедуры инсталляции файлы будут установлены, но в системной базе данных RPM сведения о них не появятся. Такие неверно работающие SRPM-пакеты являются "минным полем" и их следует избегать.

19.2.4.6. Предоставляйте контрольные суммы пакетов

Сопровождайте бинарные файлы (архивы, RPM и др.) контрольными суммами. Они позволяют пользователям проверить, не повреждены ли файлы при загрузке и не содержат ли они код "троянского коня".

Хотя существует несколько команд, которые можно использовать с данной целью (такие как sum и cksum), лучше применить криптографически безопасную хэш-функцию. Пакет GPG предоставляет данную возможность посредством параметра --detach-sign; аналогичную работу выполняет GNU-команда md5sum.

Для каждого поставляемого бинарного пакета Web-сайт проекта должен содержать контрольную сумму и команду, которая требуется для ее генерации.

19.2.5. Практические приемы хорошей коммуникации

Программа и документация не сделают мир лучше, если никто, кроме разработчика, не знает об их существовании. Разработка визуального присутствия проекта в Internet будет способствовать привлечению пользователей и других разработчиков. Ниже описаны стандартные способы реализации данной идеи.

19.2.5.1. Публикация на сайте Freshmeat

Проект можно анонсировать на сайте Freshmeat <http://www.freshmeat.net>. Кроме того что данный сайт читают широкие круги заинтересованных лиц, группа проекта является крупным источником информации для Web-каналов технических новостей.

Не надейтесь, что аудитория читает объявления о выходе новых версий программы с момента ее появления. Всегда включайте как минимум одну строку, описывающую функции программы. Пример неудачного объявления: "Announcing the latest release of FooEditor, now with themes and ten times faster" (Анонсируется последняя версия FooEditor, теперь с темами и работает в десять раз быстрее). Удачное объявление: "Announcing the latest release of FooEditor, the scriptable editor for touch-typists, now with themes and ten times faster" (Анонсируется последняя версия FooEditor, редактора для профессиональных наборщиков, который можно использовать в сценариях, теперь с темами и работает в десять раз быстрее).

19.2.5.2. Публикация в соответствующих группах новостей

Найдите группу новостей Usenet, непосредственно относящуюся к разработанному приложению, и анонсируйте в ней проект. Анонсировать проект следует только там, где

функция кода будет уместной, и проявлять при этом сдержанность.

Если (например) выпускается версия программы, написанной на Perl, которая опрашивает IMAP-серверы, то определенно о программе следует объявить в группе comp.mail.imap. Однако, вероятно, не следует отправлять объявление в группу comp.lang.perl, если программа не является также полезным примером передовых Perl-технологий.

Объявление должно включать в себя URL-адрес Web-сайта проекта.

19.2.5.3. Создайте Web-сайт

Если разработчик намеревается создать вокруг своего проекта прочное сообщество пользователей или разработчиков, то должен существовать Web-сайт проекта. Рассмотрим стандартные разделы сайта проекта.

• Хартия проекта (почему он существует, целевая аудитория и др.).

• Ссылки для загрузки исходного кода проекта.

• Инструкции о том, как подключиться к списку (или спискам) рассылки проекта.

• FAQ (список часто задаваемых вопросов и ответов на них).

• HTML-версии документации проекта.

• Ссылки на родственные и/или конкурирующие проекты.

Примеры хорошо организованных сайтов проектов приведены в главе 16.

Простым способом создания Web-сайта является размещение проекта на одном из узлов, который специализируется на предоставлении бесплатного хостинга. В 2003 году двумя наиболее важными из таких узлов были SourceForge (который является демонстрационным и тестовым сайтом для частного сотрудничества) и Savannah (поддерживающий проекты с открытым исходным кодом на идеологической основе).

19.2.5.4. Поддерживайте списки рассылки проекта

Стандартной практикой является наличие частного списка разработчиков, посредством которого участники проекта могут общаться и обмениваться исправлениями кода. Кроме того, можно создать список оповещений для людей, которые хотят быть в курсе развития проекта.

Если проект называется "foo", то список рассылки для разработчиков может называться foo-dev или foo-friends.

Важным решением является то, насколько закрытым будет "частный" список рассылки для разработчиков. Широкое участие в обсуждении конструкции проекта часто является полезным, однако если список относительно открыт, то рано или поздно

будут возникать вопросы пользователей-новичков. Способов разрешения данной проблемы множество. Требование в документации, относящееся к новым пользователям, не задавать элементарных вопросов, не решит проблему. Так или иначе, необходимо подкреплять данное требование.

Список рассылки оповещений необходимо четко контролировать. Трафик не должен превышать нескольких сообщений в месяц. Вся цель такого списка заключается в оповещении людей, которые хотят знать о важных событиях, но не нуждаются в повседневных подробностях. Большинство таких пользователей быстро откажутся от рассылки, если она начнет значительно засорять их почтовые ящики.

19.2.5.5. Публикуйте проект в главных архивах

Отдельные крупные архивные сайты для программ с открытым исходным кодом перечислены в разделе "Поиск открытого исходного кода" главы 16. Рекомендуется размещать в них свои программы.

Ниже приводится два других важных узла.

• Сайт Сообщества Python-программистов <http://www.python.org> (Python Software Activity, для программного обеспечения, написанного на языке Python).

• CPAN или Comprehensive Perl Archive Network (полный сетевой архив Perl-программ) <http://language.perl.com/CPAN> (для программ, написанных на Perl).

19.3. Логика лицензирования: как выбрать лицензию

Выбор лицензионного соглашения предполагает решение о том, какие ограничения, если они есть, налагаются автором на использование созданного им программного обеспечения.

Если разработчик вообще не хочет ограничивать использование своей программы, то ему следует сделать ее всеобщим достоянием. В таком случае в начало каждого файла уместно вставить подобный текст.

Placed in public domain by J. Random Hacker, 2003. Share

and enjoy! (Всеобщее достояние, Дж. Рэндом Хакер, 2003.

Используйте и наслаждайтесь!)

Это означает отказ от авторского права. Любой человек может использовать данную программу или ее часть по собственному усмотрению. Большей свободы не существует.

Однако существует очень немного программного обеспечения с открытым исходным кодом, которое является всеобщим достоянием. Некоторые разработчики открытого исходного кода хотят использовать право собственности на код, чтобы гарантировать, что программа остается открытой (они склоняются к принятию лицензии GPL). Другие просто хотят контролировать свою правовую незащищенность; одним из пунктов, который входит в состав

всех лицензий открытого исходного кода, является отказ от гарантии.

19.4. Почему следует использовать стандартную лицензию

Широко известные лицензии, которые согласуются с Определением открытого исходного кода (Open Source Definition), имеют твердо установившиеся "толковательные" традиции. Разработчики (и пользователи, в той мере, как это их интересует) хорошо знают, что именно они подразумевают, и имеют обоснованное мнение о риске и компромиссах. Поэтому, если

возможно, следует использовать одну из стандартных лицензий, опубликованных на сайте OSI.

Если требуется написать собственную лицензию, то ее следует сертифицировать в OSI. Это позволит избежать множества споров и издержек. Те, кто не сталкивался с лицензионными спорами, не предполагают, насколько отвратительными они могут быть; люди становятся необузданными, потому что лицензии считаются почти священными обязательствами, которые затрагивают фундаментальные ценности сообщества открытого исходного кода.

Более того, присутствие прочной "толковательной" традиции может оказаться важным, если лицензия когда-нибудь будет проверяться в суде. Во время написания данной книги (середина 2003 года) не существовало прецедентного права поддержки или аннулирования какой-либо лицензии открытого исходного кода. Однако существует юридическая система (по крайней мере, в Соединенных Штатах и, возможно, в других странах общего права, таких как Англия и остальная часть Британского государства), предполагающая, что суды интерпретируют лицензии и контракты согласно ожиданиям и практике сообщества, в котором они возникли. Таким образом, имеется веская причина надеяться, что практика сообщества открытого исходного кода будет определяющей, когда судебная система наконец будет вынуждена вступить в действие.

19.5. Многообразие лицензий на открытый исходный код

19.5.1. Лицензия MIT или Консорциума X

Самым свободным видом лицензии на свободное программное обеспечение является тот, который гарантирует неограниченные права на копирование, использование, модификацию и распространение модифицированных копий, пока во всех модифицированных версиях содержится копия авторского права и лицензионные соглашения. Но если разработчик принимает данную лицензию, то он отказывается от права на судебное преследование кураторов.

Шаблон стандартной лицензии Консорциума X можно найти на сайте OSI <http://www.opensource.org/licenses/mit-license.html>.

19.5.2. Классическая BSD-лицензия

Следующий наименее ограничивающий вид лицензии гарантирует неограниченные права на копирование, использование, модификацию и распространение модифицированных копий, пока во всех модифицированных версиях содержится копия авторского права и лицензионные соглашения, а также в рекламу или документацию, связанную с пакетом, включены соответствующие уведомления. Обладатель лицензии должен оказаться от права судебного преследования кураторов.

Оригинальная BSD-лицензия — наиболее известная лицензия данного вида. Среди частей культуры свободного программного обеспечения, которые происходят из BSD Unix, данная

лицензия используется даже во многих свободных программах, написанных за тысячи миль от Беркли.

Нередко встречаются также второстепенные варианты BSD-лицензии, которые изменяют правообладателя и опускают требования о рекламе (что фактически делает данную лицензию эквивалентной лицензии MIT). Следует отметить, что в середине 1999 года Комитет по экспорту технологий (Office of Technology Transfer) Калифорнийского университета аннулировал статью о рекламе в BSD-лицензии. Поэтому лицензия на программное обеспечение BSD была облегчена именно таким способом. В случае использования BSD-подхода настоятельно рекомендуется применять новую лицензию (без статьи о рекламе). Данное ограничение было исключено, так как оно приводило к значительным юридическим и процедурным сложностям при определении составляющих рекламы.

Шаблон BSD-лицензии можно найти на сайте OSI <http://www.opensource.org/licenses/bsd-license.html>.

19.5.3. Артистическая лицензия

Следующий наиболее ограничивающий вид лицензии гарантирует неограниченные права на копирование, использование и локальное изменение. Данная лицензия позволяет распространять модифицированные бинарные файлы, но ограничивает распространение модифицированного исходного кода в целях защиты интересов авторов и сообщества свободного программного обеспечения.

Лицензией такого вида является Артистическая лицензия (Artistic License), которая была разработана для Perl и широко используется в сообществе Perl-разработчиков. Она требует, чтобы модифицированные файлы содержали "заметное уведомление" (prominent notice) о том, что они были изменены. Кроме того, лицензия требует, чтобы люди, распространяющие изменения, предоставили к ним свободный доступ и приложили все усилия для их передачи сообществу свободного программного обеспечения.

Копия Артистической лицензии доступна на странице <http://www.opensource.org/licenses/artistic-license.html>.

19.5.4. General Public License

Общедоступная лицензия GNU (GPL и ее производные: Library (библиотечная) или "Lesser" (облегченная) GPL) является единственной наиболее широко используемой лицензией на свободное программное обеспечение. Как и Артистическая лицензия, она позволяет распространение модифицированного исходного кода при условии, что измененные файлы содержат "заметное уведомление".

GPL требует, чтобы любая программа, содержащая защищенные данной лицензией части, полностью подчинялась GPL. (Точные обстоятельства, инициировавшие данное требование, полностью не ясны никому.)

Данные дополнительные требования фактически делают GPL более ограничивающей, чем любая другая из широко используемых лицензий. (Ларри Уолл (Larry Wall) разработал Артистическую лицензию, для того чтобы избежать данных ограничений, одновременно

добиваясь множества тех же целей.)

Ссылка на GPL и инструкции по ее применению приведены на сайте авторского права FSF <http://www.gnu.org/copyleft.html>.

19.5.5. Mozilla Public License

Mozilla Public License (Общественная лицензия Mozilla) поддерживает программное обеспечение, которое поставляется с открытым исходным кодом, но может быть связано с закрытыми модулями или расширениями. Она требует, чтобы распространяемое программное обеспечение ("Covered Code" — защищенный код) оставалось открытым, но запрещает оставлять закрытыми расширения, вызываемые через определенный API-интерфейс.

Шаблон для MPL находится на сайте OSI <http://www.opensource.org/licenses/MPL-1.1.html>.

20

Будущее: опасности и перспективы

Наилучший путь предсказать будущее — создать его.

Фраза на собрании в XEROX PARC в 1971 году —Алан Кей (Alan Key)

История не окончена. Unix продолжает расти и развиваться. Сообщество и традиции вокруг операционной системы Unix продолжают развиваться. Пытаться прогнозировать будущее рискованно, но можно ускорить его приход двумя способами: во-первых, изучая то, как операционная система Unix справилась со сложностями конструкции в прошлом, во-вторых, идентифицируя проблемы, которые требуют решения, и возможности, ожидающие использования.

20.1. Сущность и случайность в традиции Unix

Для того чтобы понять, как проектирование в Unix может измениться в будущем, следует начать с рассмотрения того, как стиль Unix программирования изменялся со временем в прошлом. Эта попытка приводит непосредственно к одной из трудностей в понимании Unix-стиля — разделение случайности и сущности. То есть опознавание тех характерных черт, которые возникают из быстротечных технических обстоятельств, и тех особенностей, которые сильно связаны с центральной сложностью Unix-проектирования — как правильно использовать модульность и абстракцию, одновременно сохраняя системы прозрачными и простыми.

Подобное различие может быть трудным, поскольку характерные особенности, которые

возникли случайно, иногда обнаруживают существенную пользу. В качестве примера рассмотрим правило "молчание — золото" в проектировании Unix-интерфейса, которое рассматривалось в главе 11; оно родилось как способ адаптации к медленным телетайпам, однако сохранилось впоследствии, поскольку программы с лаконичным выводом можно было проще комбинировать в сценариях. В настоящее время в среде, где обычной практикой является запуск нескольких программ в визуальном режиме посредством GUI-интерфейса, проявляется еще одно полезное свойство: немногословные программы не отвлекают понапрасну внимание пользователя.

Напротив, некоторые характерные черты, которые однажды казались существенными для Unix, оказались случайностями, связанными с определенным набором соотношений издержек. Например, предпочтительные в старой школе Unix программные конструкции (и мини-языки, такие как

awk(1)), осуществлявшие построчную обработку входного потока или обрабатывавшие двоичные файлы последовательно от записи к записи, в любом контексте, который необходимо было поддерживать между фрагментами сложного кода конечных автоматов. С другой стороны, Unix-проектирование новой школы, как правило, использует предположение о том, что программа может считывать весь ввод в память, а впоследствии при необходимости использовать произвольный доступ к этим данным. Действительно, современные Unix-системы предоставляют вызов

mmap(2) , который позволяет программисту отображать весь файл на виртуальную память и полностью скрывать сериализацию ввода-вывода с дисковым накопителем.

Это изменение позволяет отказаться от экономии дискового пространства, а взамен получить более простой и прозрачный код, т.е. изменяется соотношение понижающейся стоимости памяти и стоимости времени программиста. Множество отличий между конструкциями старой школы Unix в 70-х и 80-х годах прошлого века и конструкциями новой школы после 1990 года можно связать с огромным сдвигом в относительной стоимости. В результате этого сдвига в настоящее время все аппаратные ресурсы становятся на несколько порядков дешевле по отношению к времени программиста, чем это было в 1969 году.

Оглядываясь назад, можно установить три специфических технологических изменения, которые повлекли значительные перемены в стиле Unix-проектирования: межсетевой обмен, растровые графические дисплеи и персональные компьютеры. В каждом случае традиция Unix приспосабливалась к трудностям, отказываясь от тех случайностей, к которым более невозможно было адаптироваться без новых способов применения своих основных идей. Биологическая эволюция движется в том же направлении. У эволюционных биологов есть правило: "Не предполагать, что историческое происхождение определяет современную пользу или наоборот". Кратко рассматривая то, как Unix приспособилась в каждом из описанных случаев, можно заметить некоторые подсказки относительно того, как Unix может адаптироваться к будущим, еще непредвиденным технологическим переменам.

В главе 2 описана первая из этих перемен: возникновение межсетевого обмена с точки зрения истории культуры. В главе 2 рассказывается, как протокол TCP/IP связал в одно целое исходную культуру Unix и культуру сети ARPANET после 1980 года. В главе 7 материал, описывающий устаревшие методы IPC и сетевые методы, такие как System V STREAMS, указывает на то, как много было заблуждений, оплошностей и тупиковых ветвей, которые захватывали Unix-разработчиков большую часть последующего десятилетия. Было много путаницы, связанной с протоколами[148], сетевым взаимодействием машин и взаимным обменом данными между процессами на одной машине.

В конце концов, путаница исчезла, когда протокол TCP/IP победил, и BSD-сокеты заново утвердили важнейшую метафору Unix: "Все является потоком байтов". Стало нормой

использовать BSD-сокеты как для IPC, так и для сетевого взаимодействия. Более ранние методы в обеих областях почти совершенно вышли из употребления, и программное обеспечение Unix развивалось все более индифферентно относительно того, расположены ли обменивающиеся данными компоненты на одной машине или на разных машинах. Логическим результатом этого было создание World Wide Web в 1990–1991 годах.

Когда в 1984 году через несколько лет после TCP IP появилась растровая графика и пример Macintosh, возникла еще более сложная проблема. Исходные GUI-интерфейсы от Xerox PARK и Apple были замечательными, однако они связывали вместе слишком много уровней системы для того, чтобы Unix программисты чувствовали себя комфортно с такой конструкцией. Немедленным ответом Unix-программистов стал явный принцип отделения политики от механизма. Система X Window была представлена к 1988 году. Они отделили наборы полезных компонентов X от диспетчера дисплея, который выполнял низкоуровневую обработку графики. Благодаря этому была создана архитектура, которая была модульной и четкой в понятиях Unix, а также позволяла с течением времени проще развивать улучшения в политике.

Однако это была простая часть проблемы. Сложной ее частью было решение — должна ли вообще Unix иметь унифицированную политику интерфейса, и если да, то какой она должна быть. Несколько различных попыток представить ее посредством частных инструментальных наборов (таких как Motif) провалились. В настоящее время в этой области конкурируют инструментарии GTK и Qt. Несмотря на то, что дебаты по этому вопросу не прекратились, постоянство различных стилей пользовательского интерфейса, рассмотренных в главе 11, впечатляет. Unix-проектирование новой школы сохранило командную строку и справилось с напряжением между подходами GUI и CLI благодаря связыванию большого количества пар CLI-ядро/GUI-интерфейс, которые могут использоваться в обоих стилях.

Как технология, персональный компьютер представил несколько главных проблем проектирования. Процессоры 386-й серии и более поздние версии были достаточно мощными для того, чтобы предоставить системам, разработанным на их основе, соотношение затрат, подобное соотношению, характерному для мини-компьютеров, рабочих станций и серверов, на которых сформировалась операционная система Unix. Истинной трудностью было изменение потенциального рынка для Unix- систем; более низкая общая стоимость аппаратного обеспечения сделала персональные компьютеры привлекательными для чрезвычайно широкой и менее технически искушенной категории пользователей.

Поставщики частных Unix-систем, привыкшие к большей прибыли от продажи более мощных систем опытным покупателям, никогда не интересовались этим рынком. Первые серьезные инициативы по направлению к настольным системам конечных пользователей исходили от сообщества открытого исходного кода и были восприняты в основном по идеологическим причинам. Согласно аналитическим исследованиям рынка, по состоянию на середину 2003 года операционная система Linux заняла около 4-5% этого рынка, что вполне сопоставимо с объемами Apple Macintosh.

Независимо от того покажет ли Linux когда-либо более высокие результаты, ответ Unix-сообщества уже ясен. Об этом уже говорилось в главе 3, при рассмотрении вопроса о заимствовании нескольких технологий (таких как XML) из других систем и натурализации GUI-интерфейсов в Unix-мире. Однако основное внимание все-таки уделяется модульности и четкому коду — созданию инфраструктуры для серьезных высоконадежных вычислений и коммуникаций.

Этот акцент подтверждается историей крупномасштабных проектов, подобных Mozilla и OpenOffice.org, которые стартовали в конце 90-х годов. В обоих указанных случаях наиболее важной темой в отклике сообщества было отнюдь не требование новых функций или соблюдение сроков поставки. Главной идеей в сообществе была неприязнь к гигантским

монолитам и общее понимание того, что прежде чем эти большие программы перестанут быть препятствием, их придется облегчить, подвергнуть рефакторингу и разделить на модули.

Вопреки большому числу инноваций, сопутствовавших данным технологиям, отклики на все эти три технологии были консервативными относительно фундаментальных правил Unix-проектирования — модульности, прозрачности, отделения политики от механизма и других качеств, которые рассматривались ранее в настоящей книге. Мудрым решением Unix-программистов было возвращение к первоначальным принципам, т.е. попытке получить больший выигрыш преимущественно из основных абстракций Unix — потоков, пространства имен и процессов, а не из иерархии новых абстракций.

20.2. Plan 9: каким представлялось будущее Unix

Известно, как обычно представляется будущее Unix. Оно было определено исследовательской группой Bell Labs, которая построила Unix, в работе под названием "Plan 9 from Bell Labs"[149]. Операционная система Plan 9 представляла собой попытку воссоздать Unix и сделать ее лучше.

Главной проблемой проектирования, которую конструкторы попытались разрешить в операционной системе Plan 9, была интеграция графики и повсеместного использования сети в комфортабельной Unix-подобной структуре. Они придерживались выбора Unix, организовывая промежуточный доступ к любому возможному количеству системных служб посредством единого, большого иерархического пространства имен файлов. Фактически они его улучшили. Многие средства, которые в Unix были доступны посредством различных узкоспециальных интерфейсов, подобных BSD-сокетам,

fcntl(2) и

ioctl(2) , в операционной системе Plan 9 были доступны посредством обычных операций чтения и записи в специальные файлы аналогичные файлам устройств. Для обеспечения переносимости и простого доступа почти все интерфейсы устройств были текстовыми, а не двоичными. Большинство системных служб (включая, например, систему оконного интерфейса) представляли собой

файловые серверы , содержащие специальные файлы или деревья каталогов, представляющие обслуживаемые ресурсы. Представляя все ресурсы в виде файлов, операционная система Plan 9 превратила проблему доступа к ресурсам, расположенным на различных серверах, в проблему доступа к файлам на различных серверах.

В операционной системе Plan 9 файловая модель, еще больше соответствующая духу Unix, чем модель самой Unix, была объединена с новой идеей: частным пространством имен. Каждый пользователь (а, по сути, каждый процесс) могли иметь собственное представление системных служб путем создания собственного дерева точек монтирования файловых серверов. Некоторые точки монтирования файловых серверов устанавливаются вручную пользователем, а другие автоматически устанавливаются во время регистрации пользователя в системе. Поэтому (как указано в обзорной статье

"Plan 9 from Bell Labs") "/dev/cons всегда ссылается на терминальное устройство, а /bin/date на корректную версию команды date, однако определение файлов, которые должны быть представлены этими именами, зависит от различных обстоятельств, таких как архитектура машины, выполняющей команду date".

Наиболее важная особенность операционной системы Plan 9 заключается в том, что все подключенные файловые серверы предоставляют одинаковый интерфейс, подобный файловой системе, независимо от скрытой за ними реализации. Некоторые из них могут соответствовать локальным файловым системам, некоторые — удаленным файловым системам, доступ к которым происходит по сети, некоторые могут соответствовать экземплярам системных серверов, запущенных в пользовательском пространстве (например, система оконного интерфейса или альтернативный набор сетевых протоколов), а некоторые могут соответствовать интерфейсам ядра. Для пользователей и клиентских программ все описанные случаи выглядят одинаково.

В обзорной статье о Plan 9 представлен способ, с помощью которого реализован FTP-доступ к удаленным узлам. В операционной системе Plan 9 не существует команды

ftp(1) . Вместо нее используется файловый сервер

ftpfs , а каждое FTP-соединение выглядит как точка монтирования файловой системы. Сервер

ftpfs автоматически преобразовывает команды открытия, чтения и записи файлов и каталогов в точке монтирования в FTP-транзакции. Таким образом, все обычные инструменты обработки файлов, такие как

ls(1), mv(1) и

cp(1) , просто работают как в точке монтирования FTP, так и через границы с остальной частью пользовательского представления пространства имен. Единственное отличие состоит в том, что пользователь (или его сценарии и программа) замечают разницу в скорости получения данных.

Plan 9 обладает и другими полезными функциями, включая воссоздание некоторых из наиболее проблемных областей интерфейсов системных вызовов Unix, устранение суперпользователя и пересмотр многих других интересных функций. "Родословная" операционной системы Plan 9 безупречна, ее конструкция элегантна, и она выявляет некоторые значительные ошибки в конструкции Unix. В отличие от большинства попыток второй системы, она создала архитектуру, которая во многих аспектах проще и более элегантна, чем архитектура ее предшественницы. Почему же Plan 9 не превзошла Unix во всем мире?

Можно назвать множество специфических причин — недостаток каких-либо серьезных попыток продвижения данной системы на рынке, ограниченная документация, большая путаница и препятствия, связанные с платой и лицензионными отчислениями. Тем, кто незнаком с Plan 9, кажется, что она функционирует в основном как опытный образец для написания интересных статей по исследованию операционных систем. Однако сама Unix ранее преодолела все подобные препятствия и привлекла преданных последователей, распространивших ее по всему миру. Почему же этого не случилось с Plan 9?

Глубокий анализ данной истории мог бы, конечно, прояснить ситуацию, но в 2003 году она такова, что Plan 9 провалилась просто потому, что не стала настолько убедительным усовершенствованием Unix, чтобы заменить свою предшественницу. По сравнению с Plan 9, Unix "скрепит, гремит и имеет очевидные пятна ржавчины", однако, она выполняет свою работу достаточно хорошо для того, чтобы удерживать позиции. Это урок для честолюбивых системных архитекторов: самым опасным врагом наилучшего решения является существующая база кода, которая просто достаточно хороша.

Некоторые идеи Plan 9 были впитаны современными Unix-системами, особенно наиболее инновационными версиями с открытым исходным кодом. А во FreeBSD файловая система

/proc смоделирована в точности, как в Plan 9, и ее можно использовать для опроса или управления работающими процессами. Системные вызовы

rfork(2) в FreeBSD и

clone(2) в Linux смоделированы на основе

rfork(2) в Plan 9. Файловая система /proc в Linux, кроме информации о процессах, содержит еще и множество файлов устройств, синтезированных наподобие Plan 9, которые используются для опроса и управления внутренними параметрами ядра с помощью преимущественно текстовых интерфейсов. Экспериментальные версии Linux 2003 года реализовали точки монтирования процессов, что является серьезным шагом в сторону частных пространств имен Plan 9. Различные Unix-системы с открытым исходным кодом двигаются в направлении общесистемной поддержки кодировки UTF-8, которая фактически была создана для Plan 9[150].

Вполне вероятно, что со временем гораздо больше функций Plan 9 будут работать в Unix, по мере того как различные части архитектуры Unix будут плавно устаревать. Это одно из возможных направлений развития будущей Unix.

20.3. Проблемы в конструкции Unix

Операционная система Plan 9 "очищает" Unix, но добавляет лишь одну новую концепцию (частное пространство имен) к ее основному набору конструктивных идей. Однако есть ли серьезные проблемы в этих базовых идеях? В главе 1 рассматривалось несколько областей, в которых Unix, вероятно, можно считать неудачной. В настоящее время, когда движение в поддержку открытого исходного кода "передало будущее" конструкции Unix обратно в руки программистов и технических специалистов, эти проблемы близки к разрешению. Ниже проблемы Unix рассматриваются снова, для того чтобы лучше понять, как Unix будет развиваться в будущем.

20.3.1. Unix-файл представляет собой только большой блок байтов

Любой файл в Unix представляет собой только большой блок байтов без каких-либо других атрибутов. В частности, не существует возможности сохранять за пределами данных файла информацию о его типе или указатель на связанную прикладную программу.

В более широком смысле все рассматривается как поток байтов. Даже аппаратные устройства являются потоками байтов. Данная метафора была большим успехом ранней Unix и большим преимуществом в мире, где (например) компилируемые программы не могли осуществлять вывод, который мог бы быть отправлен обратно компилятору. Из данной метафоры выросли каналы и shell-программирование.

Однако метафора байтового потока в Unix

настолько существенна, что в Unix имеются проблемы при интеграции программных объектов с операциями, которые не вписываются в байтовый поток или файловый состав операций (создание, открытие, чтение, запись, удаление). Это особенно является проблемой для GUI-объектов, таких как пиктограммы, окна и "активные" документы. Внутри классической

Unix- модели мира единственный способ расширить существующую метафору байтового потока заключается в использовании вызовов ioctl, печально известных как уродливая коллекция лазеек в пространство ядра.

Приверженцы операционных систем семейства Macintosh склонны громогласно критиковать это. Они пропагандируют модель, в которой одно имя файла может иметь как "ветвь" данных, так и "ветвь" ресурсов. Ветвь данных соответствует байтовому потоку в Unix, а "ветвь" ресурсов является семейством пар имя/значение. Приверженцы Unix предпочитают такие подходы, которые делают данные файла самоописательными, так чтобы фактически те же метаданные хранились внутри файла.

Проблема Unix-подхода состоит в том, что каждая программа, которая записывает файл, должна иметь информацию о его структуре. Таким образом, например, если требуется, чтобы информация о типе файла хранилась внутри данного файла, то каждое инструментальное средство, обрабатывающее данный файл, должно "позаботиться" о том, чтобы либо сохранить поле типа неизменным, либо интерпретировать его, а затем перезаписать. Несмотря на то, что организовать это теоретически возможно, на практике такие конструкции были бы слишком хрупкими.

С другой стороны, поддержка файловых атрибутов сопряжена с трудными вопросами о том, какие файловые операции должны их сохранять. Очевидно, что при копировании именованного файла в другой именованный файл должны также копироваться атрибуты файла источника, как и его данные. Однако что произойдет, если файл был перенаправлен в новый файл с помощью команды

cat(1) ?

Ответ на этот вопрос зависит от того, что подразумевается под атрибутами — они действительно представляют собой свойства имен файлов, или они некоторым волшебным способом встроены в данные файла как разновидность неотображаемого заголовка или окончания файла. Какие операции сделают свойства отображаемыми?

Разработчики файловой системы Xerox PARC боролись с данной проблемой в 70-х годах прошлого века. Для конструкции был характерен "открытый сериализованный" вызов, который возвращал байтовый поток, содержащий как атрибуты, так и содержимое файла. Если вызов применялся к каталогу, то он возвращал сериализацию атрибутов каталога плюс сериализацию всех файлов, хранящихся в нем. Не заметно, чтобы этот подход когда-либо был усовершенствован.

Ядро 2.5 Linux уже поддерживает присоединение подобных пар имя/значение как свойств имени файла, однако на момент написания данной книги эта возможность еще широко не использовалась в приложениях. Последние версии операционной системы Solaris имеют приблизительно эквивалентную функцию.

20.3.2. Слабая поддержка GUI-интерфейсов в Unix

Опыт Unix доказывает, что использование нескольких метафор в качестве базиса для интегрирующей структуры представляет собой мощную стратегию (обсуждение интегрирующих структур и общего контекста приведено в главе 13). Визуальная метафора "в сердце" современных GUI-интерфейсов (файлы, представленные пиктограммами и открываемые щелчком мыши, который запускает некоторую программу, предназначенную для обработки и обычно способную создавать и редактировать данные файлы) с момента ее

разработки в Xerox PARC в 1970-х годах доказала свою способность успешно и долговременно удерживать пользователей и разработчиков интерфейсов.

Несмотря на значительные недавние усилия, в 2003 году Unix все еще поддерживала эту метафору слабо и неохотно — существовало множество уровней, несколько соглашений и только слабые конструкционные утилиты. Типичная реакция со стороны опытных профессионалов Unix — подозревать, что это отражает более глубокие проблемы с самой GUI-метафорой.

Я думаю, что часть проблемы заключается в том, что мы до сих пор не имеем правильной метафоры. Например, в Macintosh, для того чтобы удалить файл, я перетаскиваю его в корзину, но когда я перетаскиваю его на диск, то файл копируется, кроме того, я не могу перетащить файл на пиктограмму принтера, для того чтобы распечатать его, потому что для этого используется меню. Я мог бы продолжать. Это похоже на файлы в OS/360, до того как появилась Unix с ее простой (но не слишком простой) файловой идеей. Стив Джонсон.

В главе 11 приводились цитаты Брайана Кернигана и Майка Леска по этому же поводу, но проблему невозможно решить, лишь обвиняя GUI-интерфейс. Несмотря на все его недостатки, имеется огромная потребность в GUI-интерфейсах со стороны конечных пользователей. Предположим, что удалось создать правильную метафору на уровне взаимодействия с пользователем. Была бы в таком случае Unix способна изящно поддерживать GUI-интерфейс?

Вероятно, нет. Авторы касались данной проблемы, анализируя, является ли модель представления файлов в виде большого блока байтов адекватной. Для более развитой поддержки GUI-интерфейсов механизм мог бы быть предоставлен файловыми атрибутами в стиле Macintosh. Однако кажется маловероятным, что такой подход полностью разрешит проблему. Объектная модель в Unix не включает в себя верных фундаментальных конструкций. Необходимо определить, какой в действительности была бы строгая интегрирующая структура для GUI интерфейсов. И, что не менее важно, как ее можно интегрировать с существующими структурами Unix. Это сложная проблема, требующая фундаментального понимания, которое пока невозможно "среди шума и путаницы" программной инженерии и академических исследований.

20.3.3. Удаление файлов в Unix необратимо

Пользователи с опытом работы в системе VMS или те, кто помнит TOPS-20, часто испытывают недостаток средств для контроля версий файлов, которые были характерны для этих систем. Открытие существующего файла для записи или удаления фактически приводило к его переименованию предсказуемым способом, а новое имя включало в себя номер версии. И только явная операция удаления файла версии фактически стирала данные.

Unix обходится без этого, немалой ценой раздражения пользователей, когда удаляются не те файлы из-за опечатки или неожиданного влияния групповых символов оболочки.

Вероятнее всего, в ближайшем будущем эта проблема не решится на уровне операционной системы. Разработчикам Unix нравятся четкие простые операции, которые выполняют именно то, что требует пользователь, даже если пользовательские инструкции могут приводить к дестабилизации системы. Инстинкт диктует им, что защита пользователя от его самого должна осуществляться на уровне графического интерфейса или на уровне приложения, но не на уровне операционной системы.

20.3.4. Unix предполагает статичную файловую систему

Unix с одной стороны имеет весьма статичную модель мира. Неявно предполагается, что программы выполняются недолго, так что содержимое файлов и каталогов во время их выполнения можно считать статичным. Не существует стандартного, хорошо организованного способа запрашивать у системы уведомление для приложения в случае, если определенный файл или каталог изменяются. Это становится значительной проблемой при написании долго работающей программы пользовательского интерфейса, которой необходимы сведения об изменении ее окружения.

В операционной системе Linux предусмотрена функция уведомления об изменениях файлов и каталогов[151], и эти функции скопированы в некоторых версиях BSD, однако они еще не перенесены на другие Unix системы.

20.3.5. Конструкция системы управления задачами была плохо реализована

Не считая возможности приостанавливать процессы (что само по себе является тривиальным дополнением к планировщику, который мог бы быть сделан довольно безопасно), управление задачами предусмотрено для переключения терминала между несколькими процессами. К сожалению, при этом решается простейшая часть проблемы — определение нажатия клавиши. Вместе с тем сложные части, такие как сохранение и восстановление состояния экрана, передаются приложению.

Действительно хорошая реализация данного средства была бы полностью невидимой для пользовательских процессов: без выделенных сигналов, без необходимости сохранять и восстанавливать режимы терминалов, без необходимости для приложений перерисовывать экран в случайные промежутки времени. Моделью должна быть виртуальная клавиатура, которая иногда подключается к реальной (и блокирует ее, если пользователь запрашивает ввод, когда она не подключена), а также виртуальный экран, который время от времени становится видимым на реальном экране (и может блокировать, а может не блокировать вывод, когда он невидимый), с системой, выполняющей мультиплексирование подобно мультиплексированию доступа к диску, процессору и другим ресурсам, но не влияющей на пользовательские программы в целом[152].

При правильной реализации потребовалось бы, чтобы tty-драйвер Unix не просто поддерживал буфер линии, но и полностью отслеживал текущее состояние экрана. Кроме того, потребовалось бы, чтобы сведения о типах терминалов были известны на уровне ядра (возможно, с помощью процесса демона), для того чтобы оно могло соответствующим образом выполнить восстановление, когда приостановленный процесс снова переводится в приоритетный режим. Последствия неправильной реализации заключаются в том, что ядро не способно отключить сеанс как задачу

xterm или

Emacs от одного терминала и подключить его к другому терминалу (тип которого мог бы отличаться).

Поскольку использование Unix сместилось в сторону X-дисплеев и эмуляторов терминалов,

управление задачами стало сравнительно менее важным и этот вопрос уже не имеет прежней остроты. Однако удручает тот факт, что до сих пор не существует функции приостановления/подключения/отключения. Данная функция могла бы быть полезной для сохранения состояния терминальных сеансов между сеансами регистрации в системе.

Широко распространенная программа с открытым исходным кодом, которая называется

screen(1), решает некоторые из этих проблем[153]. Однако поскольку пользователь должен ее вызвать явно, не гарантируется, что ее возможности будут присутствовать в каждом терминальном сеансе. Кроме того, код уровня ядра, который перекрывает ее в функциональной части, не был удален.

20.3.6. В Unix API не используются исключительные ситуации

Язык C испытывает недостаток средств восстановления для обработки именованных исключительных ситуаций со связанными данными[154]. Таким образом, C-функции в Unix API сообщают об ошибках путем возвращения известного значения (обычно -1 или указатель на NULL-символ) и установки глобальной переменной errno.

Оглядываясь назад, становится ясно, что это является источником многих неочевидных ошибок. Программисты в спешке часто пренебрегают проверкой возвращаемых значений. Так как исключительные ситуации не обрабатываются, нарушается правило исправности; процесс выполнения программы продолжается до тех пор, пока позднее ошибка не проявится при выполнении как некоторый сбой или повреждение данных.

Отсутствие исключений также означает, что некоторые задачи, которые должны были бы быть простыми идиомами — как выход из обработчика сигналов по версии с сигналами Беркли-стиля — должны осуществляться с помощью сложного кода, который является причиной проблем переносимости и чреват ошибками.

Данная проблема может быть скрыта (и обычно скрывается) привязками Unix API в таких языках, как Python или Java, в которых есть исключительные ситуации.

Недостаток исключений фактически является индикатором проблемы с последующими более крупными последствиями. Слабая онтология типов в C делает проблематичным обмен данными между реализованными на нем языками более высокого уровня. Например, большинство современных языков имеют списки и словари как первичные типы данных. Однако, поскольку они не имеют канонического представления в C, попытки передавать списки между (например) Perl и Python являются неестественными и требуют большого количества связующего кода.

Существуют технологии, такие как CORBA, которые решают более крупную проблему, но они тяжеловесны и склонны задействовать большое количество преобразований во время выполнения.

20.3.7. Вызовы

ioctl(2) и

fcntl(2) являются препятствиями

Механизмы

ioctl(2) и

fcntl(2) обеспечивают способ написания перехватчиков (hooks) в драйверах устройств. Первоначальным историческим использованием

ioctl(2) была установка параметров, таких как скорость передачи и количество фреймирующих битов в драйверах последовательных линий, отсюда и название ("I/O control"). Позднее вызовы ioctl были добавлены для других драйверов, а

fcntl(2) был добавлен как перехватчик в файловую систему.

С годами вызовы ioctl и fcntl распространились. Они часто слабо документированы, а также нередко являются источником проблем переносимости. Каждому из них сопутствует неаккуратное нагромождение макроопределений, описывающих типы операций и специальные значения аргументов.

Основная проблема в данном случае та же, что и "большой блок байтов"; объектная модель Unix слабая и не оставляет естественного пространства для размещения многих вспомогательных операций. Разработчики получают сложный выбор из неудовлетворительных альтернатив. Вызовы fcntl/ioctl проходят через устройства в /dev, новые специализированные системные вызовы или методы через специализированные виртуальные файловые системы, которые привязаны к ядру (например, /proc Linux и др.).

Пока не ясно, улучшится ли в будущем объектная модель Unix, а если улучшится, то каким образом. Если MacOS-подобные атрибуты файлов станут обычной функцией Unix, подстройка "магических" именованных атрибутов на драйверах устройств может взять на себя роль, которую в настоящее время играют вызовы ioctl/fcntl (это, по крайней мере, исключило бы необходимость в применении макроопределений перед использованием интерфейса). Выше уже отмечалось, что операционная система Plan 9, в которой вместо модели файл/поток байтов используется именованный файловый сервер или файловая система как базовый объект, представляет другой возможный путь.

20.3.8. Модель безопасности Unix, возможно, слишком примитивна

Возможно, полномочия пользователя root слишком широки, и в Unix должны быть возможности более четкой градации полномочий или ACL (Access Control Lists — списки контроля доступа) для функций системного администрирования, чем один суперпользователь, который может все. Люди, которые принимают данную позицию, спорят, что многие системные программы имеют постоянные root- привилегии благодаря механизму setuid. Если даже одну из них можно будет взломать, то вторжения последуют везде.

Однако это довольно слабый аргумент. Современные Unix-системы позволяют включать учетную запись любого пользователя в несколько групп. Используя полномочия на выполнение, а также установку битов идентификатора группы на выполняемые файлы программ, можно заставить каждую группу функционировать в качестве ACL-списка для файлов или программ.

Однако данная теоретическая возможность используется очень мало, и это наводит на мысль, что на практике потребность в ACL-списках является гораздо меньшей, чем в теории.

20.3.9. Unix имеет слишком много различных видов имен

Unix объединила файлы и локальные устройства — они являются просто потоками байтов. Однако сетевые устройства, доступные через сокеты, имеют иную семантику и другое пространство имен. Plan 9 показывает, что файлы вполне можно сочетать как с локальными, так и с удаленными (сетевыми) устройствами, и все это может управляться через пространство имен, которое способно динамически настраиваться для каждого пользователя и даже для каждой программы.

20.3.10. Файловые системы могут считаться вредными

С конца 1970-х годов продолжается захватывающая история исследований в области постоянных хранилищ объектов и операционных систем, которые не имеют общей глобальной файловой системы вообще, а трактуют дисковые накопители как огромную область подкачки и выполняют все операции посредством визуализированных объектных указателей.

Современные исследования в данном направлении (такие как EROS[155]) подтверждают, что такие конструкции могут предоставлять большие преимущества, включая доказуемое соответствие политике безопасности и более высокую производительность. Однако следует отметить, что если это проигрыш Unix, то это равный проигрыш всех ее конкурентов. Ни одна крупная действующая операционная система еще не предпочла направление EROS[156].

20.3.11. На пути к глобальному адресному пространству Internet

Возможно, URL-адреса недостаточно успешны. Оставим последнее слово о возможных направлениях будущего развития Unix за создателем этой системы.

Моим идеалом будущего является развитие удаленного интерфейса файловой системы (а ля Plan 9), а затем его реализация в Internet как стандарта вместо HTML. Это было бы действительно здорово. Кен Томпсон.

20.4. Проблемы в окружении Unix

Культура старой Unix почти совершенно воссоздана в движении открытого исходного кода. Это сохраняет нас от вырождения, но это также означает, что проблемы открытого исходного кода в настоящее время являются также нашими проблемами.

Одна из таких проблем — как сделать разработку открытого исходного кода экономически целесообразной? Мы вновь "вернулись к нашим корням" в общем, открытом процессе ранней эпохи Unix. Мы почти совершенно выиграли технический спор об отказе от секретности и

частного управления. Мы продумали способы сотрудничества с рынками и менеджерами на более равноправных условиях, чем это было возможно в 70-х и 80-х годах прошлого века, и во многом наши эксперименты были успешными. В 2003 году Unix-системы с открытым исходным кодом и их основные группы разработчиков достигли такого авторитета в обществе, который не так давно, еще в середине 90-х годов был бы невообразимым.

Пройден долгий путь, но такой же долгий путь еще предстоит пройти. Мы знаем, какие бизнес-модели могут работать в теории, и сейчас мы можем даже указать отдельные успехи, которые демонстрируют работу этих моделей на практике. Теперь мы должны показать, что они могут надежно работать в течение долгого времени.

Это не обязательно будет легкий переходный период. Открытый исходный код превратил программное обеспечение в сервисную индустрию. Фирмы-провайдеры услуг (вспомним о медицинской и юридической практике) не могут расширяться путем вливания большого капитала. А те, которые пытаются это сделать, только увеличивают свои фиксированные затраты, превышают свою доходную базу и отмирают. Выбор сводится к тому, чтобы получать плату за советы или в качестве пожертвований, основать мелкий, низкозатратный сервисный бизнес или найти богатого патрона (какую-нибудь крупную фирму, которая нуждается в использовании и модификации программного обеспечения с открытым исходным кодом в целях своего бизнеса).

В целом, можно ожидать, что денежные затраты на найм разработчиков программного обеспечения будут расти по тем же причинам, по которым и почасовая оплата механиков растет по мере снижения цен на автомобили[157]. Однако контролировать такие траты для любого индивидуума или фирмы будет все труднее. Будет много состоятельных программистов, но меньше миллионеров. Фактически это признак прогресса, неспособности вырваться из системы. Но это повлечет крупные изменения на рынке, а возможно, это будет означать, что инвесторы потеряют тот небольшой интерес, который они проявляли при финансировании новых программных проектов.

Одной важной подзадачей, связанной с возрастающей трудностью поддержки действительно крупных программных предприятий, является организация тестирования силами конечных пользователей. Исторически концентрация Unix-культуры на инфраструктуре означала, что мы не стремились создавать программы, ценность которых зависела от предоставления комфортного интерфейса для конечных пользователей. В настоящее время, особенно в Unix-системах с открытым исходным кодом, которые стремятся конкурировать непосредственно с Microsoft и Apple, ситуация изменяется. Однако пользовательские интерфейсы должны систематически тестироваться реальными конечными пользователями — и в этом отношении наблюдаются некоторые трудности.

Тестирование со стороны реальных конечных пользователей требует средств, специалистов и уровня мониторинга, которые трудно достижимы для распределенных групп добровольцев, характерных для разработки открытого исходного кода. Следовательно, может оказаться, что текстовые процессоры, электронные таблицы и другие "продуктивные" приложения придется оставить в руках крупных, поддерживаемых корпорациями проектов, таких как OpenOffice.org, которые могут позволить себе такие издержки. Разработчики открытого исходного кода считают единые корпорации основной причиной затруднений и беспокоятся о такой зависимости, однако лучшее решение пока не найдено.

Все эти проблемы экономические. Существуют и другие проблемы, более "политические", поскольку успех создает врагов.

Некоторые из них известны. Стремление Microsoft к непотопляемой монополии на компьютерные вычисления в середине 1980-х годов сделало поражение Unix стратегической целью компании (т.е. еще за пять лет до того, как мы узнали, что втянуты в борьбу). В

середине 2003 года, несмотря на то, что несколько развивающихся рынков, на которые рассчитывала корпорация, были почти полностью захвачены системой Linux, Microsoft являлась богатейшей и наиболее мощной программной компанией в мире. Microsoft хорошо известно — для того чтобы выжить, ей необходимо победить новые Unix-системы, созданные внутри движения открытого исходного кода, а для того чтобы сделать это, она должна разрушить или дискредитировать создавшую их культуру.

Возвращение Unix в сообщество открытого исходного кода и ее связь со свободной культурой Internet также создали новых врагов. Голливуд и крупные медиа- корпорации чувствуют угрозу со стороны Internet и инициировали многовекторную атаку на неконтролируемую разработку программного обеспечения. Существующее законодательство, такое как Digital Millennium Copyright Act (Закон об охране авторских прав в цифровом тысячелетии), уже не раз использовалось для судебного преследования разработчиков, которые создавали программы, неугодные медиа-магнатам (самые известные случаи, несомненно, связаны с программным обеспечением DeCSS, которое позволяет проигрывать зашифрованные DVD-диски). Уже продуманы схемы, подобные так называемому "Trusted Computing Platform Alliance" (альянс достоверных вычислительных платформ) и "Palladium threaten"[158], чтобы сделать разработку открытого исходного кода фактически незаконной. И если открытый исходный код придет в упадок, то Unix, весьма вероятно, придет в упадок вместе с ним.

Unix, хакеры и Internet против Microsoft, Голливуда и медиа-гигантов. Это борьба, которую необходимо выиграть по всем традиционным причинам профессионализма, преданности ремеслу и взаимной преданности сообществу. Однако существуют и более весомые причины этой борьбы. Возможности политики все более формируются коммуникационными технологиями — сильнее сегодня тот, кто может использовать их, кто может подвергать их цензуре, кто может их контролировать. Правительственный и корпоративный контроль над содержимым сетей, а также тем, что люди могут делать с помощью своих компьютеров, является ощутимой долгосрочной угрозой политической свободе. Кошмарным сценарием представляется ситуация, когда корпоративный монополизм и политики, непрерывно ищущие власти, как постоянные естественные союзники, объединятся друг с другом и создадут подоплеку для возрастающего регулирования, репрессий и криминализации цифрового общения. Противостоим этому мы — воины свободы — свободы всеобщей, а не только своей собственной.

20.5. Проблемы в культуре Unix

Не менее важными, чем технические проблемы операционной системы Unix и трудности, являющиеся результатом ее успеха, являются культурные проблемы окружающего ее сообщества. Существует как минимум две серьезные проблемы: меньшая — сложность внутреннего развития, и большая — сложность преодоления нашего "исторически сложившегося аристократического высокомерия".

Меньшая трудность — трение между гуру старой Unix-школы и приверженцами открытого исходного кода новой школы. Успех Linux, в частности, не является полностью удобным феноменом для многих старых Unix-программистов. Частично это проблема поколения. Сильная энергия, наивный и веселый фанатизм поколения Linux иногда раздражает старейшин, переживших 70-е годы и (часто справедливо) считающих себя мудрее. Тот факт, что дети добиваются успеха там, где отцы терпят неудачу, только обостряет данную проблему.

Более значительная психологическая проблема стала очевидной автору только после трех

дней, проведенных на конференции Macintosh-разработчиков в 2000 году. Погружение в культуру программирования с диаметрально противоположными предположениями было весьма поучительным.

Все Macintosh-программисты "вращаются" вокруг пользовательского опыта. Они — архитекторы и декораторы. Они проектируют снаружи внутрь, в первую очередь задаваясь вопросом: "Какой вид взаимодействия необходимо поддерживать?", а затем на заднем плане строят прикладную логику для удовлетворения запросов конструкции пользовательского интерфейса. Это приводит к появлению очень красивых программ и слабой, неустойчивой инфраструктуры. В одном печально известном примере MacOS 9 диспетчер памяти иногда требовал от пользователя высвобождения памяти вручную путем остановки программы (также вручную), которая уже завершила работу, но все еще остается резидентной. Приверженцы Unix внутренне протестуют против такой ошибочной конструкции; они не понимают, как пользователи Macintosh могут с этим мириться.

Напротив, Unix-программисты полностью заняты инфраструктурой. Мы — водопроводчики и каменотесы. Мы проектируем изнутри, создавая могучие машины для решения абстрактно определенных проблем (например: "Как обеспечить надежную доставку пакетов из пункта A в пункт B через ненадежную аппаратуру и каналы?"). Затем мы скрываем машины за тонкими и часто крайне уродливыми интерфейсами. Команды

date(1), find(1) и

ed(1) являются широко известными примерами, но существуют еще сотни других. Приверженцы Macintosh внутренне протестуют против такой ошибочной конструкции; они не понимают, как пользователи Unix могут с этим мириться.

Обе конструкторские философии обоснованы, но два лагеря имеют большие сложности при рассмотрении противоположных точек зрения. Типичный рефлекс Unix-разработчика — выбрасывать из головы Macintosh-программы как кричащую массу, приятные для глаза дилетанта картинки, и продолжать создавать программное обеспечение, которое притягивает внимание других Unix-разработчиков. И если пользователям программа не нравится — тем хуже для пользователей; они вернутся, когда им откроется тайна.

Во многом такая ограниченность хорошо нам послужила. Мы — хранители Internet и World Wide Web. Наше программное обеспечение и наши традиции господствуют в серьезных вычислениях, приложениях, где обязательными требованиями являются надежность двадцать четыре часа в сутки и семь дней в неделю, а также минимальные простои. Мы действительно чрезвычайно хорошо создаем прочную инфраструктуру; мы ни в коем случае не идеальны, но не существует другой культуры создания программного обеспечения, которая приблизилась бы к нашим рекордам, и мы гордимся своей культурой.

Проблема в том, что мы все больше сталкиваемся с трудностями, которые требуют более широкого взгляда. Большинство компьютеров в мире находятся не в серверных комнатах, а скорее в руках тех самых конечных пользователей. В ранние дни Unix, до персональных компьютеров, наша культура определяла себя частично как протест против культа мэйнфреймов, хранителей "большого железа". Позднее мы впитали идеализм ранних энтузиастов мини-компьютеров: мощные компьютеры — людям. Но сегодня

мы являемся служителями культа;

мы люди, которые поддерживают сети и "большое железо". И наше скрытое требование гласит: "Если вы хотите использовать наши программы, то вы должны научиться думать, как мы".

В 2003 году в нашей идеологии наметилось глубокое противоречие — внутренний конфликт

между высокомерием и миссионерским популизмом. Мы хотим переубедить и переделать 92% пользователей, для которых компьютеры означают игры, мультимедиа, блестящие GUI-интерфейсы и (что касается наиболее технических пользователей) легкие почтовые программы, текстовые процессоры и электронные таблицы. Мы затрачиваем значительные усилия на проекты, подобные GNOME и KDE, предназначенные для того, чтобы придать Unix симпатичное лицо. Но в сердце мы до сих пор высокомерны, глубоко не желаем, а во многих случаях не способны идентифицировать или выслушивать пожелания неискушенных пользователей.

Для нетехнических пользователей создаваемые нами программы часто кажутся либо непонятными и приводят в замешательство, либо грубыми и помпезными. Даже если мы пытаемся укрепить дружественность к пользователю, мы прискорбно непоследовательны в этом. Во многих случаях наше отношение и рефлексы, унаследованные от старой школы Unix, просто неприемлемы для данной задачи. Даже когда мы хотим выслушать неискушенного пользователя и помочь ему, мы не знаем, как это сделать — мы проецируем на него свои категории и понятия и предоставляем ему "решения", которые он находит такими же пугающими, как сама проблема.

Величайшая наша проблема как культуры — сможем ли мы перерасти предположения, которые так хорошо нам служили, сможем ли мы признать не только интеллектуально, но и силой повседневной практики, что Macintosh-разработчиков можно понять? Их точка зрения в более широком смысле, но менее характерно для Mac описана в

"The Inmates Are Running the Asylum" [14], проницательной и логичной книге о том, что ее автор называет

дизайном взаимодействия , содержащей (несмотря на случайные причуды) много тяжелой правды, которую следовало бы знать каждому Unix-программисту.

Мы можем уклониться от этого; мы можем остаться служителями культа, привлекая избранное меньшинство лучших и талантливейших, образованную элиту, сфокусированную на нашей исторической роли хранителей программной инфраструктуры и сетей. Но в таком случае наша культура, весьма вероятно, придет в упадок и со временем потеряет динамизм, который поддерживал нас в течение десятилетий. Кто-то другой будет служить людям; кто-то другой придет туда, где будут мощности и деньги, и будет править будущим 92% всего программного обеспечения. Есть вероятность, независимо от того, будет ли этим кто-то именно Microsoft, что они будут делать это с помощью практик и программ, которые нам не очень нравятся.

Или мы можем действительно принять вызов. Движение открытого исходного кода упорно старается сделать это. Однако непрерывной работы и интеллекта, который мы привнесли в решение других проблем в прошлом, недостаточно. Необходимо принципиально изменить наши позиции.

В главе 4 подчеркивалась, как важно отбросить ограничивающие предположения и отказаться от прошлого в решении технических проблем, предлагая параллели с идеями Дзэн об освобождении и "разуме начинающего". Сейчас нам приходится работать над более серьезным видом освобождения. Мы должны научиться терпимости к нетехническим пользователям и освободиться от некоторых давних предубеждений, которые сделали нас такими успешными в прошлом.

Примечательно, что культура Macintosh начала объединяться с нашей — в основе MacOS X стоит Unix, и сегодня Mac-разработчики (хотя и в некоторых случаях не без трудностей) приспосабливают свой разум к изучению достоинств Unix, сфокусированных на инфраструктуре. Аналогично, нашей проблемой будет принять достоинства Macintosh,

связанные с ориентацией системы на пользователя.

Имеются также другие признаки того, что Unix-культура избавляется от своей изолированности. Одним из них является слияние, которое, по всей видимости, будет продолжаться, сообщества Unix/открытый исходный код с движением, которое называется "гибкое программирование" (agile programming)[159]. В главе 4 отмечалось, что Unix-программисты удачно ухватились за идею рефакторинга, одного из предубеждений мыслителей гибкого программирования. Рефакторинг и другие концепции гибкого программирования, такие как блочное тестирование (unit-testing) и дизайн вокруг областей (design around stories), кажется, ясно выражают и обостряют практические приемы, которые до этого были широко распространены в Unix-традиции, но только неявно. С другой стороны, Unix-традиция может привнести в гибкое программирование устойчивость и уроки многолетнего опыта. Поскольку открытое программное обеспечение приобретает свою долю рынка, очень вероятно, что эти культуры объединятся, как это было с ранними культурами Internet и Unix после 1980 года.

20.6. Причины верить

В будущем операционной системы Unix много проблем. Хотели бы мы действительно изменить его?

За более чем тридцатилетнюю историю мы преуспели в разрешении многих трудностей. Мы были первопроходцами лучших практических приемов программной инженерии. Мы создали сегодняшние Internet и Web. Мы создали крупнейшие, наиболее сложные и самые надежные программные системы из когда-либо существовавших. Мы пережили монополию IBM и выступили против монополии Microsoft.

Однако не все это можно назвать триумфом. В 1980-х годах мы почти разрушили свою культуру, согласившись с частным захватом Unix. Мы долго пренебрегали компьютерами низкого класса и нетехническими конечными пользователями и таким образом предоставили Microsoft удобный случай для внедрения низкокачественных стандартов программного обеспечения. Умные наблюдатели неоднократно объявляли крах нашей технологии, сообщества и наших ценностей.

Но мы всегда стремительно возвращались. Мы делаем ошибки, но мы учимся на своих ошибках. Мы передали нашу культуру через поколения; мы впитали многое из лучшего опыта ранних академических хакеров, ARPANET-экспериментаторов, энтузиастов мини-компьютеров и множества других культур. Движение открытого исходного кода возродило силу и идеализм наших ранних лет, и сегодня мы сильнее и многочисленнее, чем когда-либо.

До сих пор доводы против Unix-хакеров всегда были в краткосрочном выигрыше, но проигрывали в долгосрочной перспективе. Мы можем победить, если решимся на это.

Приложения

A

Глоссарий аббревиатур

В данном приложении даны определения наиболее важных аббревиатур, используемых в основном тексте книги.

API

Application Programming Interface. Программный интерфейс приложений. Набор процедурных вызовов, который обменивается данными со связываемой библиотекой процедур или ядром операционной системы, или их комбинацией.

BSD

Berkeley System Distribution, или

Berkeley Software Distribution. Дистрибутив системы Беркли, или дистрибутив программного обеспечения Беркли. Документальные свидетельства неоднозначны. Общее название дистрибутивов Unix, выпущенных Computer Science Research Group в Калифорнийском университете в Беркли в период между 1976 и 1994 годами, а также Unix-систем с открытым исходным кодом, генетически происходящих от данных дистрибутивов.

CLI

Command Line Interface. Интерфейс командной строки. Считается некоторыми архаичным, но остается весьма полезным в мире Unix.

CPAN

Comprehensive Perl Archive Network. Полный сетевой архив Perl-программ. Центральный Web-репозиторий Perl-модулей и расширений <http://cpan.org>.

GNU

GNU's Not Unix! GNU — не Unix! Рекурсивная аббревиатура для проекта фонда свободного программного обеспечения (Free Software Foundation) по созданию клона Unix с полностью свободным программным обеспечением. Данный проект не достиг абсолютного успеха, однако в его рамках создано множество основных инструментов современной Unix-разработки, включая редактор Emacs и коллекцию компиляторов GNU (GNU Compiler Collection).

GUI

Graphical User Interface. Графический пользовательский интерфейс. Современный стиль интерфейса приложений, использующих мышь, окна и пиктограммы, созданный в лаборатории Xerox PARC в 1970-х годах как противоположность более старым CLI- или rogue-подобным стилям.

IDE

Integrated Development Environment. Интегрированная среда разработки. Автоматизированные инструментальные средства разработки с GUI-интерфейсом для создания кода, описания средств и быстрого просмотра структур данных. В Unix данные средства не получили широкого распространения по причинам, описанным в главе 15.

IETF

Internet Engineering Task Force. Инженерная группа по решению конкретной задачи в Internet. Орган, ответственный за определение Internet-протоколов, таких как TCP/IP. Свободная, коллегиальная организация, главным образом состоящая из технических специалистов.

IPC

Inter-Process Communication. Межпроцессное взаимодействие. Любой метод передачи данных между процессами, запущенными в отдельных адресных пространствах.

MIME

Multipurpose Internet Mail Extensions. Многоцелевые расширения почтового стандарта в Internet. Ряд документов RFC, которые описывают стандарты для внедрения двоичных и многокомпонентных сообщений в почту стандарта RFC-822. Кроме использования для передачи почты, MIME-формат используется в качестве нижнего уровня для важных протоколов уровня приложений, включая HTTP и BEEP.

OO

Object Oriented. Объектно-ориентированный. Стиль программирования, при котором код и управляемые им данные инкапсулируются в изолированные (теоретически) контейнеры — объекты. В противоположность данному стилю, в необъектно-ориентированном программировании внутреннее устройство структур данных и кода отображается более свободно объектно-ориентированное.

OS

Operating System. Операционная система. Базовое программное обеспечение машины. Именно оно управляет задачами, распределяет пространство памяти и предоставляет пользователю стандартный интерфейс для управления приложениями. Средства, предоставляемые операционной системой, и ее общая философия проектирования чрезвычайно сильно оказывают влияние на стиль программирования и техническую культуру, которая формируется вокруг машин, поддерживающих данную систему.

PDF

Portable Document Format. Переносимый формат документов. PostScript-язык для управления принтерами и другими устройствами, формирующими изображение. PDF представляет собой последовательность PostScript-страниц, упакованных с аннотациями таким образом, что его можно легко использовать в качестве формата отображения.

PDP-11

Programmable Data Processor 11. Программируемый процессор данных 11. Вероятно, единственная наиболее успешная конструкция мини-компьютера в истории; непосредственный предшественник компьютеров VAX. Впервые был представлен в 1970 году, последний PDP-11 появился в 1990 году. Данный мини-компьютер был первой крупной Unix-платформой.

PNG

Portable Network Graphics. Формат переносимой сетевой графики. Стандарт Консорциума WWW (The World Wide Web Consortium), рекомендуемый формат для растровых графических изображений. Элегантно разработанный формат растровой графики, описанный в главе 5.

RFC

Request For Comment. Запрос на комментарии. Internet-стандарт. Название, которое возникло в то время, когда документы считались предложениями, представленными для рассмотрения в ходе тогда еще несуществующего, но ожидаемого формального процесса согласования. Формальный процесс согласования никогда не осуществлялся.

RPC

Remote Procedure Call. Удаленный вызов процедур. Использование IPC-методов, которые призваны создать иллюзию того, что процессы, обменивающиеся данными, работают в одном адресном пространстве и могут экономично (a) совместно использовать комплексные структуры и (b) вызывать друг друга как библиотеки функций, игнорирующие задержку и другие проблемы производительности. Общеизвестно, что эту иллюзию трудно поддерживать.

TCP/IP

Transmission Control Protocol/Internet Protocol. Протокол управления передачей/Internet-протокол. Базовый протокол Internet с момента преобразования NCP (Network Control Protocol — протокол управления сетью) в 1983 году. Обеспечивает гарантированную транспортировку потоков данных.

UDP/IP

Universal Datagram Protocol/Internet Protocol. Протокол универсальных дейтаграмм/Internet протокол. Обеспечивает негарантированную, но с малой задержкой, транспортировку небольших пакетов данных.

UI

User Interface. Пользовательский интерфейс.

VAX

Формально

Virtual Address Extension — виртуальное расширение адреса. Название классической конструкции мини-компьютеров, разработанной Digital Equipment Corporation (позднее претерпевшей слияние с Compaq, которая в свою очередь впоследствии была поглощена Hewlett-Packard) на основе PDP-11. Первый мини-компьютер VAX был представлен в 1977 году. В течение десяти лет после 1980 года мини-компьютеры VAX были в числе важнейших Unix-платформ. Доработанные микропроцессоры поставляются и в настоящее время.

Б

Список литературы

Хронология индустрии Unix и GNU/Linux опубликована на страницах <http://snap.nlc.dcccd.edu/learn/drkelly/hst-hand.htm> и <http://www.robotwisdom.com/linux/timeline.html>. Историческая диаграмма версий операционной системы Unix также представлена на Web-странице <http://www.levenez.com/Unix/>.

1. Appleton R.

Improving Context Switching Performance of Idle Tasks under Linux. 2001. - C.

188.

Статья доступна на Web-странице <http://cs.nmu.edu/~randy/Research/Papers/Scheduler/>.

2. Baldwin C., Clark K.

Design Rules, Vol 1: The Power of Modularity. MIT Press, 2000. - C.

107.

3. Bentley J.

Programming Pearls. (Второе издание). Addison-Wesley, 2000. — C.

245.

В третьем эссе данной книги "Data Structures Programs" обсуждается учебный пример, подобный примеру в главе 9. Некоторая часть книги доступна в Web <http://www.es.bell-labs.com/cm/cs/pearls/>.

4. Blaauw G. A., Brooks F. P.

Computer Architecture: Concepts and Evolution. Addison-Wesley, 1997. — C.

122, 428.

5. Bolinger D., Bronson T.

Applying RCS and SCCS. O'Reilly & Associates, 1995. - C.

398.

Данная книга — не просто "рецептурный" справочник, в ней исследуются проблемы конструкции систем управления версиями.

6. Brokken F.

C++Annotations Version, 2002. - C.

358.

Книга также доступна в Web: <http://www.icce.rug.nl/documents/cplusplus/cplusplus.html>.

7. Brooks D.

Converting a UNIX.COM Site to Windows. 2000. - C.

82.

Книга также доступна в Web: <http://www.securityoffice.net/mssec-rets/hotmail.html#_Toc491601819>.

8. Brooks F. P.

The Mythical Man-Month, (двадцатое юбилейное издание). Addison-Wesley, 1995. - C.

39, 40, 334, 472.

9. Boehm H.

Advantages and Disadvantages of Conservative Garbage Collection. — C.

352.

Исчерпывающий анализ компромиссов между средами "с уборкой мусора" и без нее. Статья доступна в Web <http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html>

10. Cameron D., Rosenblatt B., Raymond E.

Learning GNU Emacs, (второе издание). O'Reilly & Associates, 1996. - C.

382.

11. Cannon L. W., Elliot R. A., Kirchhoff L. W., Miller J. A., Milner J. M„ Mitzw R. W., Schan E. P., Whittington N. O., Spencer H., Keppel D., Brader M.

Recommended C Style and Coding Standards, 1990. - C.

443.

Обновленная версия документа

Indian Hill C Style and Coding Standards с изменениями, внесенными тремя последними авторами. В документе описан рекомендованный стандарт кодирования для С-программ. Документ доступен в Web <http://www.apocalypse.org/pub/u/paul/docs/cstyle/cstyle.htm>.

12. Christensen C.

The Innovator's Dilemma. HarperBusiness, 2000. — C.

75.

Книга, в которой вводится понятие "пробивной технологии". Интересное исследование, демонстрирующее, как и почему солидные технологические компании остаются позади начинающих фирм. Бизнес-книга, которую следует прочесть техническими специалистам.

13. Cooper A.

The Inmates Are Running the Asylum. Sams, 1999. — C.

509.

Данная книга представляет собой глубокий анализ ошибок в конструкции программных интерфейсов и методов их исправления.

14. Comer D. Статья "Pervasive Unix: Cause for Celebration" в журнале

Unix Review, октябрь 1985. - C. 57.

15. Coram T., Lee J.

Experiences — A Pattern Language for User Interface Design, 1996. — C.

295.

Статья доступна в Web <http://www.maplefish.com/todd/papers/Experiences.html>.

16. DuBois P.

Software Portability with Imake. O'Reilly & Associates, 1993. — C.

394.

17. Eckel B.

Thinking in Java, (Третье издание). Prentice-Hall, 2003. — C.

372.

Книга доступна в Web <http://www.mindview.net/Books/TIJ/>.

18. Feller J., Fitzgerald B.

Understanding Open Source Software. Addison-Wesley, 2002. —

C. 471.

19. Flanagan D

. Java in a Nutshell. O'Reilly & Associates, 1997. - C.

372.

20. Flanagan D

. JavaScript: The Definitive Guide, (Четвертое издание). O'Reilly & Associates, 2002. - C.

235.

21. Fowler M.

Refactoring. Addison-Wesley, 1999. - C.

118.

22. Friedl J.

Mastering Regular Expressions, (Второе издание). O'Reilly & Associates,

— C. 217.

23. Miller B., Koski D., Lee C. P., Maganty V., Murthy R., Natarajan A., Steidl J.

Fuzz Revisited: A Re-examination of the Reliability of Unix Utilities and Services, 2000. — C

. 31.

Документ доступен в Web: <http://www.opensource.org/advocacy/fuzz-revisited.pdf>.

24. Gamma E., Helm R., Johnson R., Vlissides J.

Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1997. — C

. 23.

25. Gabriel R.

Good News, Bad News, and How to Win Big, 1990. - C.

328, 471.

Статья доступна в Web <http://www.dreamsongs.com/WorseIsBetter.html>.

26. Gancarz M.

The Unix Philosophy. Digital Press, 1995. - C.

19.

27. Garfinkel S., Weise D., Strassman S.

The Unix Hater's Handbook. IDG Books, 1994. —

C. 28.

Книга доступна в Web <http://research.microsoft.com/~daniel/Unix- haters.html>.

28. Gentner D., Nielsen J. "The Anti-Mac Interface". Статья в журнале

Communications of the ACM Ассоциации вычислительной техники (Association for Computing Machinery). - август 1996. - C.

290.

<http://www.acm.org/cacm/AUG96/antimac.htm>

29. Gettys J.

The Two-Edged Sword, 1998. - C.

33.

Статья доступна в Web: <http://freshmeat.net/articles/view/122/>.

30. Glickstein B.

Writing GNU Emacs Extensions. O'Reilly & Associates, 1997. - C.

374.

31. Graham P.

A Plan for Spam. - C.

246.

Статья доступна в Web: <http://www.paulgraham.com/spam.html>.

32. Harold E. R., Means W. S.

XML in a Nutshell, (Второе издание). O'Reilly & Associates, 2002. - C.

142, 467.

33. Hatton L. "Re-examining the Defect-Density versus Component Size Distribution".

IEEE Software. — март/апрель 1997. — C.

110.

Статья доступна в Web
<http://www.cs.ukc.ac.uk/people/staff/lh8/pubs/pubis697/Ubend_IS697.pdf.gz>.

34. Hatton L. "Does OO Sync with the Way We Think?".

IEEE Software, 15(3). - C.

357.

Статья доступна в Web
<http://www.cs.ukc.ac.uk/people/staff/lh8/pubs/pubis698/00_IS698.pdf.gz>.

35. Hauben R.

History of UNIX. - C.

57.

Документ доступен в Web <http://www.dei.isep.ipp.pt/docs/Unix.html>.

36. Heller S. C++:

A Dialog: Programming with the C++

Standard Library. Prentice-Hall,

2003. - C.

358.

37. Hunt A., Thomas D.

The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, 2000. - C.

18, 114.

38. Kernighan B.

Experience with Tcl/Tk for Scientific and Engineering Visualization. Доклады конференции USENIX Association Tcl/Tk Workshop, 1995. — C.

366.

Документ доступен в Web
<http://www.usenix.org/publications/library/proceedings/tcl95/full_papers/kernighan.txt>.

39. Kernighan B., Pike R.

The Unix Programming Environment. Prentice-Hall, 1984. — C.

19, 354, 360, 386.

40. Kernighan B., Pike R.

The Practice of Programming. Addison-Wesley, 1999. — C.

18, 443. (Б. Керниган, Р. Пайк. Практика программирования. ИД "Вильямс", 2004 г.)

Отличный трактат по написанию высококачественных программ, данная книга, несомненно, должна стать классикой в своей области.

41. Kernighan B., Plauger P.J.

Software Tools. Addison-Wesley, 1976. - C

. 37.

42. Kernighan B., Ritchie D.

The C Programming Language, (Второе издание). Prentice-Hall Software Series, - C.

118, 355, 428, 430.

43. Lampson B. "Hints for Computer System Design". Обзор операционных систем (ACM Operating Systems Review) Ассоциации вычислительной техники (Association for Computing Machinery). — октябрь 1983 г. — C.

53.

Документ доступен в Web <http://research.microsoft.com/~lampson/33-Hints/WebPage.html>.

44. Lapin J. E.

Portable C and Unix Systems Programming. Prentice-Hall, 1987. — C.

430.

45. Leonard A.

BSD Unix: Power to the People, from the Code, 2000. - C.

58.

Документ доступен в Web <http://dir.salon.com/tech/fsp/2000/05/16/chapter_2_part_one/index.html>.

46. Levy S.

Hackers: Heroes of the Computer Revolution. Anchor/Doubleday, 1984. — C.

67.

Книга доступна в Web <http://www.Stanford.edu/group/mmdd/Silicon-Valley/Levy/Hackers.1984.book/contents.html>.

47. Lewine D.

POSIX Programmer's Guide: Writing Portable Unix Programs. O'Reilly & Associates, 1992. - C.

434.

48. Libes D., Ressler S.

Life with Unix. Prentice-Hall, 1989. - C.

58.

В данной книге более подробно описывается ранняя история Unix, особенно период 1979-1986 годов.

49. Lions J.

Lions's Commentary on Unix, 6th Edition. Peer-To-Peer Communications, 1996. - C.

57.

PostScript-версии оригинальной книги Лайонза путешествуют по Web, данная ссылка может быть нестабильной <http://www.upl.cs.wisc.edu/~epaulson/lionc.ps>.

50. Loukides M., Oram A.

Programming with GNU Software. O'Reilly & Associates, 1996.-C

.380, 387.

51. Lutz M.

Programming Python. O'Reilly & Associates, 1996. - C.

368.

52. McIlroy M. D., Pinson E. N., Tague B. A. "Unix Time-Sharing System Forward". Статья в

техническом журнале корпорации Bell System (The Bell System Technical Journal), Bell Laboratories, 1978. — C.

34.

53. McIlroy M. D. "Unix on My Mind". В материалах конференции

Virginia Computer Users Conference. — C. 55.

54. Miller G. "The Magical Number Seven, Plus or Minus Two".

The Psychological Review. — C. 112.

Статья доступна в Web <http://www.well.com/user/smalin/miller.html>.

55. Mumon.

The Gateless Gate. — C.

179.

Книга доступна в Web <http://www.ibiblio.org/zen/cgi-bin/koan-index.pl>.

56. Ockman S., DiBona C.

Open Sources: Voices from the Open Source Revolution. O'Reilly & Associates, 1999. - C.

85.

Книга доступна в Web <http://www.oreilly.com/catalog/opensources/book/toe.html>.

57. Oram A., Talbot S.

Managing Projects with Make. O'Reilly & Associates, 1991. - C.

387.

58. Ousterhout J.

Tcl and the Tk Toolkit. Addison-Wesley, 1994.

59. Ousterhout J.

Why Threads Are a Bad Idea (for most purposes), 1996.

Доклад на конференции USENIX 1996. Соответствующего напечатанного документа нет, но презентация доступна в Web <http://home.pacbell.net/ouster/threads.pdf">.

60. Padlipsky M.

The Elements of Networking Style. iUniverse.com, 2000. - C.

128, 496.

61. David P. L. "On the Criteria to Be Used in Decomposing Systems into Modules".

Communications of the ACM. — C. 109.

Статья доступна в Web в разделе классики на странице ACM
<http://www.acm.org/classics/may96/>

62. Pike R.

Notes on Programming in C. — C.

34.

Документ доступен на странице <http://www.lysator.liu.se/c/pikestyle.html>.

63. Prechelt L.

An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a
Search/String-Processing Program. — C. 353.
<http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2000/5>.

64. Raskin J.

The Humane Interface. Addison-Wesley, 2000. — C.

282.

Краткий конспект приведен на странице
<http://humane.sourceforge.net/humane_interface/summary_of_thi.html>.

65.

The Memory Management Reference. — C.

46.

Доступно в Web <http://www.memorymanagement.org/>.

66.

The New Hacker's Dictionary, (Третье издание). MIT Press, 1996. — С.

69, 332.

Книга доступна в Web на странице файла жаргона <http://www.catb.org/~esr/jargon>.

67. Raymond E. S.

The Cathedral and the Bazaar, (Второе издание). O'Reilly & Associates, 1999. — С.

72, 471, 508.

68. Reps P., Senzaki N.

Zen Flesh, Zen Bones. Shambhala Publications, 1994. — С.

18. Превосходная антология основных Дзэн-источников как они есть.

69. Ritchie D. M.

The Evolution of the Unix Time-Sharing System, 1979. — С.

53.

Книга доступна в Web <http://cm.bell-labs.com/cm/cs/who/dmr/hist.html>.

70. Ritchie D. M.

The Development of the C Language, 1993. - С.

429.

Статья доступна в Web <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>.

71. Ritchie D. M.

An Incomplete History of the QED Text Editor, 2003. - С.

335.

Статья доступна в Web <http://cm.bell-labs.com/cm/cs/who/dmr/qed.html>.

72. Ritchie D. M., Thompson K.

The Unix Time-Sharing System. — С. 56.

Книга доступна в Web <http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html>.

73. Saltzer J. H., Reed D. P., Clark D. D. "End-to-End Arguments in System Design". Статья

ACM Transactions on Computer Systems Ассоциации вычислительной техники (Association for Computing Machinery). — ноябрь 1984 г. — С.

149.

Статья доступна в Web

<http://Web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf>.

74. Salus P. H.

A Quarter-Century of Unix. Addison-Wesley, 1994. - C.

34.

Превосходный обзор истории Unix, объясняющий многие из конструкторских решений словами их создателей.

75. Schaffer E., Wolf M.

The Unix Shell as a Fourth-Generation Language, 1991. — C.

190.

Документ доступен в Web <http://www.rdb.com/lib/4gl.pdf>. Реализация с открытым исходным кодом,

NoSQL, доступна и легко находится с помощью поисковых машин.

76. Schwartz R., Phoenix T.

Learning Perl, (Третье издание). O'Reilly & Associates, 2001. - C.

363.

77. Spinellis D. "Notable Design Patterns for Domain-Specific Languages".

Journal of Systems and Software. — февраль 2001 г. — C.

236.

Статья доступна в Web
<http://www.dmst.aueb.rg/dds/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html>.

78. Stallman R. M.

The GNU Manifesto. - C.

62.

Документ доступен в Web <http://www.gnu.org/gnu/manifesto.html>.

79. Stephenson N.

In the Beginning Was the Command Line. 1999. — C.

290.

Документ доступен в Web <http://www.cryptonomicon.com/beginning.html>.

80. Stevens W. R.

Unix Network Programming. Prentice-Hall, 1990. - C.

205.

Классическая книга по данной тематике. В некоторых поздних изданиях опущено рассмотрение сетевых средств Unix версии 6, таких как mx().

81. Stevens W. R.

Advanced Programming in the Unix Environment. Addison-Wesley, 1992. - C.

18.

Исчерпывающее руководство по Unix API. Полезная книга для опытного программиста или талантливого новичка, стоящее дополнение к

Unix Network Programming

82. Stroustrup B.

The C++ Programming Language. Addison-Wesley, 1991. - C.

358.

83. Tanenbaum A. S., Van Renesse R.

A Critique of the Remote Procedure Call Paradigm. Доклады конференции EUTECO'88, 1988. - C.

207.

84. Tidwell D.

XSLT:MasteringXML Transformations. O'Reilly & Associates, 2001. - C.

223.

85. Torvalds L., Diamond D

.Just for Fun: The Story of an Accidental Revolutionary. HarperBusiness, 2001. - C.

428, 436.

86. Vaughan G. V., Tromey T., Taylor I. L.

GNU Autoconf, Automake, and Libtool. New Riders Publishing, 2000. - C.

395.

Пользовательское руководство по инструментам автоконфигурирования GNU. Доступно в Web <http://sources.redhat.com/autobook/>.

87. Vo K.-P. "The Discipline and Method Architecture for Reusable Libraries".

Software Practice & Experience, 2000. — C.

124.

Статья доступна в Web <http://www.research.att.com/sw/tools/vcodex/dm-spe.ps>.

88. Wall L., Christiansen T., Orwant J.

Programming Perl, (Третье издание). O'Reilly & Associates, 2000. - C.

363.

89. Welch B.

Practical Programming in Tcl and Tk. Prentice-Hall, 1999. - C.

366.

90. Williams S.

Free as in Freedom. O'Reilly & Associates, 2002. - C.

70.

Книга доступна в Web <http://www.oreilly.com/openbook/freedom/index.html>.

91. Yourdon E.

Death March. The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects. Prentice-Hall, 1997. - C.

408.

B

Персональный вклад

Любой из посетивших конференцию USENIX в модном отеле может сказать, что фраза вроде: "Вы один из тех компьютерных людей, не так ли?" подобна фразе: "Смотри, вот еще одна поразительно подвижная форма омерзительного пудинга!", прозвучавшей из уст гостиничной официантки, разносящей коктейли. —Элизабет Звикки (Elizabeth Zwicky)

Кен Арнольд (Ken Arnold) был членом группы, которая создавала версии 4BSD Unix. Он написал первичную библиотеку

curses(3) и был одним из авторов оригинальной игры

rogue(6) . Кен Арнольд является также соавтором

Java Reference Manual (Справочник по языку Java) и одним из ведущих экспертов по Java и OO-технологиям.

Стивен M. Белловин (Steven M. Bellovin), работая в 1979 г. в Университете Северной Каролины, вместе с Томом Траскоттом (Tom Truscott) и Джимом Эллисом (Jim Ellis) создал Usenet. В 1982 году он вошел в состав AT&T Bell Laboratories, где одним из первых начал заниматься исследованием вопросов безопасности, криптографии и сетей на основе Unix-систем, а также участвовал в других проектах. Он является активным членом IETF (Internet Engineering Task Force — Инженерная группа по решению конкретной задачи в Internet), а также Национальной инженерной академии (National Academy of Engineering).

Стюарт Фельдман (Stuart Feldman) был одним из членов группы разработчиков Unix в Bell Labs. Он написал утилиты

make(1) и

f77(1) . В данный момент он является вице-президентом IBM, отвечающим за компьютерные исследования.

Джим Геттис (Jim Gettys) вместе с Бобом Шейфлером (Bob Scheifler) и Китом Паккардом (Keith Packard) был одним из разработчиков системы X Window в конце 1980-х годов. Он написал X-библиотеки, X-лицензию, а также провозгласил основное кредо в конструкции X: "механизм, а не политика".

Стив Джонсон (Steve Johnson) написал утилиту

yacc(1), а затем использовал ее для написания портативного компилятора C, который заменил оригинальный DMR C и стал предшественником большинства более поздних C-компиляторов в Unix.

Брайан Керниган (Brian Kernighan) является единственным видным выразителем хорошего стиля в сообществе Unix. Он стал автором и соавтором нескольких книг, ставших классикой, включая "

The Practice of Programming", "The C Programming Language", "The Unix Programming Environment" . Работая в Bell Labs, он стал соавтором языка

awk(1) и ведущим разработчиком семейства инструментов troff, включая

eqn(1) (совместно с Лориндой Черри (Lorinda Cherry

)), pic(1), и

grap(1) (с Джоном Бентли (Jon Bentley)).

Дэвид Корн (David Korn) написал командный интерпретатор Korn, стилистический предшественник почти всех современных оболочек в Unix. Он создал UWIN, эмулятор Unix для тех, кто вынужден использовать Windows. Дэвид также провел исследования по разработке файловых систем и средств для обеспечения переносимости исходного кода.

Майк Леск (Mike Lesk) входил в состав первоначальной команды разработчиков Unix в Bell Labs. Среди прочего его вкладом является пакет макросов

ms, инструменты для обработки текстов

tb(1) и

refer(1) , генератор лексических анализаторов

lex(1) и программа UUCP (программа копирования из Unix в Unix).

Дуг Макилрой (Doug McIlroy) возглавлял исследовательскую группу Bell Labs, где возникла Unix, и создал Unix-канал (pipe). Он написал такие утилиты, как

spell(1), diff(1), sort(1), join(1), tr(1) , и другие классические инструменты Unix, а также способствовал определению стиля документации Unix. Он также проводил исследования по алгоритмам распределения памяти и компьютерной безопасности.

Маршал Кирк Маккьюзик (Marshall Kirk McKusick) реализовал быструю файловую систему 4.2BSD и был специалистом по компьютерным исследованиям в группе CSRG (Computer Systems Research Group — Исследовательская группа по компьютерным системам) в Беркли, следя за разработкой и выпуском версий 4.3BSD и 4.4BSD.

Кит Паккард (Keith Packard) внес главный вклад в создание первоначального кода X11. Во время начавшейся в 1999 году второй фазы разработки, Кит переписал код визуализации в X, создав более мощную и значительно более компактную реализацию, подходящую для портативных компьютеров и PDA.

Эрик С. Реймонд (Eric S. Raymond) с 1982 года создает программное обеспечение для Unix. В 1991 году он исправил

"The New Hacker's Dictionary" (Новый словарь хакера), и с тех пор изучает Unix-сообщество и культуру Internet-хакеров, с исторической и антропологической точки зрения. В 1997 году как результат этих исследований появилась книга

"The Cathedral and the Bazaar" , которая помогла (пере)определить и оптимизировать движение открытого исходного кода. Он лично курирует более 30 программных проектов и около 10 ключевых FAQ-документов.

Генри Спенсер (Henry Spencer) был лидером первой волны программистов, принявших Unix (средина 1970-х годов). Его вклад: свободно-распространяемая библиотека

getops(3) , первая библиотека с открытым исходным кодом для обработки строк, модуль с открытым исходным кодом для обработки регулярных выражений, который нашел свое применение в 4.4BSD и как исходная точка стандарта POSIX. Кроме того, Спенсер — известный эксперт в области языка C, он является соавтором журнала C News. В течение многих лет он считается лидером в Usenet и одним из наиболее уважаемых представителей сообщества.

Кен Томпсон (Ken Thompson) — создатель Unix.

Г

Корни без корней: Unix-коаны Мастера Фу

Предисловие редактора

Открытие коллекции коанов, известных как "

Корни без корней" (Rootless Root), вызвало ожесточенные споры в ученых кругах. Проливают ли эти аутентичные документы новый свет на методы преподавания патриархов Unix? Или это просто искусная стилизация, подразумевающая среди прочего и авторитет таких полумифических фигур, как патриархи Томпсон, Ритчи и Макилрой, для доктрин, которые эволюционировали ближе к нашему времени?

Ответить определенно на этот вопрос невозможно. Все участники дискуссии якобы имеют отношение к этой освященной веками классике

('The Tao of Programming") [160]. Но

"Корни без корней" резко отличаются по тону и стилю от свободных поэтических анекдотов в переводе Джеймса (James), сконцентрированных на загадочной и замечательной личности

Мастера Фу.

Было бы более уместно искать параллели в

"AI-коанах" [161]

(AI Koans); действительно, есть определенные признаки того, что автор трактата "

Корни без корней", возможно, редактировал некоторые версии

"AI Koans". Также будет небезосновательным поиск связей с

"Loginataka" [162]; действительно, вполне возможно, что неизвестные авторы

"AI Koans" и

"Loginataka" на самом деле являются одним и тем же лицом, возможно, учеником самого Мастера Фу.

Следует также упомянуть и о

"Сказках Дзэн-мастера Грега" (Tales of Zen Master Greg) [163], хотя существуют определенные сомнения в их древности, и поэтому маловероятно, что они повлияли на

"Корни без корней" .

Можно с достаточной уверенностью сказать, что название этой работы является ссылкой на Дзэн-классику

"Врата без врат" (Gateless Gate) [164] Мумона (Mumon). Эти отголоски чувствуются в нескольких коанах.

Возник серьезный спор о том, к какой из школ, выросших из раннего эпохального путешествия патриарха Томпсона в Беркли, следует причислить Мастера Фу — к Восточной (Нью-Джерси) или к Западной. Если этот вопрос до сих пор не разрешен, то, вероятно, именно потому, что мы не можем даже точно установить — а существовал ли вообще Мастер Фу? Под этим именем может скрываться просто группа учителей или все последователи Дхармы.

Даже если предположить, что легенда о Мастере Фу сформировалась вокруг учения какой-то одной личности, как быть с его любимым учеником Ньюби (Nubi)? У Ньюби есть все признаки типичного образцового ученика. Вспоминаются легенды о любимом последователе Будды Ананде (Ananda). Представляется вероятным, что существовал исторический Ананда, однако никакие следы его реальной личности не пережили эфемерный процесс, который отполировал личность Будды и превратил его в вечный миф.

В конечном итоге все, что мы можем, — это принять эти поучительные истории и извлечь те зерна мудрости, которые в них содержатся.

Мастер Фу и десять тысяч строк

Однажды Мастер Фу сказал заезжему программисту: "В одной строке кода shell-сценария больше духа Unix, чем в десяти тысячах строк на языке C!"

Программист, гордый своими познаниями в C, ответил: "Может ли быть такое? Ведь C —

язык, в котором реализовано само ядро Unix!"

На это Мастер Фу ответил: "Это так. Тем не менее, в одной строке shell-сценария больше духа Unix, чем в десяти тысячах строк C!"

Программист выглядел удрученным. "Но ведь через язык C мы познаем просвещенность патриарха Ритчи! Мы уподобляемся человеку с операционной системой и компьютером, который получает непревзойденную производительность!"

Мастер Фу сказал: "То, что ты говоришь, правда. Однако в одной строке shell-сценария больше духа Unix, чем в десяти тысячах строк C".

Программист усмехнулся и поднялся, чтобы удалиться. Но Мастер Фу кивнул своему ученику Ньюби, который писал строку shell-кода на стоящей рядом белой доске, и сказал: "Господин программист, посмотрите на этот конвейер! Не заняла бы его реализация на C десять тысяч строк?"

Просматривая то, что писал Ньюби, программист что-то бормотал в бороду. В конце концов, он согласился, что это так.

"И сколько часов потребовалось бы вам для реализации и отладки этой программы на языке C?"

"Много", — признал заезжий программист. "Но только безумец стал бы тратить столько времени, когда его ждет множество более достойных задач".

"Так кто лучше понимает дух Unix?" — спросил Мастер Фу. "Тот, кто пишет десять тысяч строк, или тот, кто, сознавая тщетность этих усилий, извлекает пользу, не программируя?"

Услышав это, программист достиг просветления.

Мастер Фу и Скрипт Кидди

Незнакомец из страны Вут пришел к Мастеру Фу во время его утренней трапезы.

"Я не раз слышал о вашем величии, — сказал он. — Пожалуйста, научите меня всему, что знаете".

Ученики Мастера Фу переглянулись, смущенные варварским языком пришельца. Мастер Фу только улыбнулся и ответил: "Вы хотите изучить путь Unix?"

"Я хочу быть волшебником-хакером,— ответил незнакомец, — и владеть всеми компьютерами".

"Я не учу этому", — ответил Мастер Фу.

Волнение незнакомца росло. "Отец, вы — позер и ничего больше, — сказал он. — Если бы Вы знали хоть что-нибудь, то научили бы меня".

"Есть путь, который может привести тебя к мудрости", — сказал Мастер Фу. Мастер нацарапал какой-то IP-адрес на клочке бумаги. "Взлом этого сервера не составит для тебя большого труда, поскольку его хранители не компетентны. Возвращайся и расскажи мне, что ты ищешь".

Незнакомец поклонился и вышел. Мастер Фу закончил трапезу.

Прошли дни, а затем месяцы. О незнакомце забыли.

Спустя годы незнакомец из страны Вут вернулся.

"Будь ты проклят!" — воскликнул он. — Я взломал этот сервер, это было не трудно, как ты и сказал. Но ФБР схватило меня и бросило в тюрьму".

"Хорошо", — сказал Мастер Фу. — Ты готов к следующему уроку". Он написал IP-адрес на бумаге и передал его незнакомцу.

"Вы

с ума сошли ?", — пронзительно вскрикнул тот. — После всего, что я прошел, я не собираюсь снова взламывать компьютеры!"

Мастер Фу улыбнулся. "Здесь, — сказал он,— начинается мудрость".

Услышав это, странник достиг просветления.

Мастер Фу рассуждает о двух дорогах

Мастер Фу учил своих студентов:

"В учении дхармы есть направление, выражаемое мантрой патриарха Макилроя — "Делай хорошо одну вещь", которая подчеркивает, что программное обеспечение движется по пути Unix, если оно ведет себя просто и последовательно, и обладает свойствами, которые могут быть легко смоделированы в мозгу пользователя и использованы другими программами".

"Но есть и другое направление в учении дхармы, примером которого может служить великая мантра патриарха Томпсона — "Находясь в сомнении, используй грубую силу", и различные сутры о большей ценности 90% функций прямо

сейчас , чем 100%

позже , что подчеркивает надежность и простоту реализации".

"Теперь скажите мне: каким программам присущ дух Unix?"

Помолчав, Ньюби заметил: "Учитель, эти два учения могут противоречить друг другу".

"Простой реализации может не хватить логики в граничных ситуациях, таких как нехватка ресурсов или неудачная попытка закрыть окно или таймаут во время незаконченной транзакции".

"Когда возникают подобные граничные ситуации, поведение программного обеспечения становится непредсказуемым и сложным. Конечно, это не есть путь Unix".

Мастер Фу кивнул в знак согласия.

"С другой стороны, хорошо известно, что причудливые алгоритмы хрупки. Кроме того, каждая попытка охватить граничные случаи имеет тенденцию взаимодействовать с центральными алгоритмами других программ и с кодами, описывающими другие граничные ситуации".

"Таким образом, попытка изначально охватить все граничные случаи, гарантируя "простоту описания", может на деле привести к созданию кода, который излишне усложнен или слишком неустойчив, или который в случае, если он переполнен ошибками, не будет завершен никогда. Конечно, это не есть путь Unix".

Мастер Фу кивнул в знак согласия.

"Каков же, в таком случае, путь дхармы?" — спросил Ньюби.

И учитель ответил: "Когда орел летит, забывает ли он о том, что его лапы касались земли? Когда тигр после прыжка настигает свою жертву, забывает ли он о моменте, проведенном в воздухе? Три фунта VAX!"

Услышав это, Ньюби достиг просветления.

Мастер Фу и консультант по методологии

Когда Мастер Фу и его ученик Ньюби посещали святые места, по вечерам Мастер Фу имел обыкновение выступать перед неофитами Unix тех городов и сел, где они останавливались на ночлег.

Однажды среди тех, кто собрался его послушать, оказался консультант по методологии.

"Если при доводке вы не профилируете регулярно ваш код в поисках узких мест, то вы уподобляетесь рыбаку, который закидывает сеть в озеро, в котором нет рыбы", — сказал Мастер Фу.

"Не верно ли тогда и то, — сказал консультант по методологии, — что если вы не замеряете постоянно вашу производительность при управлении ресурсами, то вы уподобляетесь рыбаку, который закидывает сеть в озеро, в котором нет рыбы".

"Однажды я встретил рыбака, который только что уронил сеть в озеро, по которому плыла его лодка, — сказал Мастер Фу. — Он долго шарил по дну лодки, пытаясь найти ее".

"Но если он уронил свою сеть в озеро, — сказал консультант по методологии, — то почему он искал ее в лодке?"

"Потому, что он не умел плавать",— ответил Мастер Фу.

Услышав это, консультант достиг просветления.

Мастер Фу рассуждает о графическом пользовательском интерфейсе

Однажды вечером Мастер Фу и Ньюби посетили собрание программистов, которые встретились, чтобы поучиться друг у друга. Один из программистов спросил у Ньюби, к какой школе принадлежит он и его учитель. Когда Ньюби сказал ему, что он и его учитель — последователи Великого Пути Unix, программист презрительно усмехнулся.

"Средства командной строки Unix грубые и отсталые, — насмешливо сказал он. — Современные, правильно спроектированные операционные системы делают все через

графический интерфейс пользователя".

Мастер Фу не проронил ни слова, но указал на Луну. Находившийся поблизости пес залаял на руку учителя.

"Я не понимаю вас",— сказал программист.

Мастер Фу молчал и показал на образ Будды. Потом он указал на окно.

"Что вы хотите мне этим сказать?" — спросил программист.

Мастер Фу указал на голову программиста. Потом он указал на камень.

"Почему вы не можете сказать яснее?" — потребовал программист.

Мастер Фу задумчиво нахмурился, дважды щелкнул программиста по носу и бросил его в находившийся рядом мусорный контейнер.

Пока программист пытался выбраться из горы мусора, пес ходил рядом и лаял на него.

В этот момент программист достиг просветления.

Мастер Фу и фанатик Unix

Один фанатик Unix, услышав, что Мастер Фу обладает мудростью Великого Пути, пришел к нему поучиться. Мастер Фу сказал ему:

Когда патриарх Томпсон изобрел Unix, он не понял этого. Потом к нему пришло понимание, но он уже не мог ничего изобрести.

Когда патриарх Макилрой изобрел канал, он знал, что это преобразит программное обеспечение, но он не знал, что это изменит его мышление.

Когда патриарх Ритчи изобрел язык C, он обрек программистов на адские муки переполнения буфера, повреждения данных и ошибки из-за недействительного указателя.

Действительно, патриархи были слепы и глупы!.

Фанатик был очень рассержен словами Мастера Фу.

"Просвещенные, — запротестовал он, — открыли нам Великий путь Unix. И если мы будем насмехаться над ними, мы потеряем добродетель и возродимся как звери или MCSE".

"Бывает ли когда-либо твой код полностью без погрешностей и ошибок?" — спросил Мастер Фу?

"Нет, — ответил фанатик, — такое недоступно человеку".

"Мудрость патриархов, — сказал Мастер Фу, — в том, что они

знали , что они безумцы".

Услышав это, фанатик достиг просветления.

Мастер Фу рассуждает о природе Unix

Один ученик сказал Мастеру Фу: "Нам говорят, что фирма SCO удерживает реальную власть над Unix".

Мастер Фу кивнул в знак согласия.

Ученик продолжал: "Однако нам также говорят, что другая фирма, OpenGroup, также удерживает реальную власть над Unix".

Мастер Фу кивнул в знак согласия.

"Как такое возможно?" — спросил ученик.

Мастер Фу ответил: "SCO действительно владеет кодом Unix, но код Unix — это не сама Unix. OpenGroup действительно владеет маркой Unix, но название Unix — это не сама Unix".

"В чем же тогда сущность Unix?" — спросил студент.

Мастер Фу ответил: "Не в коде. Не в имени. Не в мышлении. Вообще ничего материального. Вечное изменение без перемен".

"Сущность Unix проста и пуста. Поскольку она проста и пуста, она сильнее тайфуна".

"Повинуясь естественным законам, она непреклонно расцветает в умах программистов, ассимилируя конструкции в свою собственную природу. Всякое программное обеспечение, которое хотело бы конкурировать с Unix, должно стать таким, как Unix: пустым, пустым, глубоко пустым, абсолютно лишенным содержания потоком!"

Услышав это, ученик достиг просветления.

Мастер Фу и конечный пользователь

В другой раз, когда Мастер Фу давал публичную лекцию, один пользователь, наслушавшись рассказов о мудрости Учителя, подошел к нему за советом.

Он трижды поклонился Мастеру Фу. "Я хочу постичь тайны Великого Пути, но командная строка вводит меня в замешательство".

Некоторые из наблюдавших это неофитов начали насмехаться над пользователем, называя его невежественным и говоря, что Великий путь Unix предназначен только для тех, в ком есть порядок и интеллект.

Учитель поднял руку, призывая к тишине, и позвал самого шумного из неофитов, который засмеялся первым, подойти к месту, где они сидели с пользователем.

"Расскажи мне" — спросил он у неофита, — о коде, который ты написал, и о работе по проектированию, которую ты проделал".

Неофит начал, заикаясь, отвечать, но не мог ничего сказать.

Мастер Фу повернулся к пользователю. "Скажи мне, — осведомился он, — зачем ты ищешь

Великий Путь?"

"Мне не нравится программное обеспечение, которое окружает меня, — отвечал пользователь. — Оно работает ненадежно и не радует глаз и сердце. Услышав о том, что путь Unix, хотя и труден, но превосходен, я пытаюсь отбросить все препоны и обман".

"И чем же ты занимаешься, если так борешься с нынешним программным обеспечением?" — спросил Мастер Фу.

"Я — строитель, — ответил пользователь. — Многие дома в этом городе построены моими руками".

Мастер Фу повернулся к неофиту. "Кошка может насмехаться над тигром, — сказал он, — но это не превратит мяуканье в рев".

Услышав это, неофит достиг просветления.

Дополнительная информация

В процессе составления оригинала данной книги какое-либо частное программное обеспечение не использовалось. Черновики были набраны из главных файлов XML-DocBook, созданных с помощью редактора

GNU Emacs. Формирование PostScript-представления осуществлялась с помощью программы Тима Bora (Tim Waugh) xmlto, таблиц стилей XSL Нормана Уолша (Norman Walsh), программы

xlsproc Даниеля Вейлларда (Daniel Veillard), PassiveTeX-макросов Себастьана Ратца (Sebastian Rahtz), TeTeX-дистрибутива наборной машины TEX Дональда Кнутта (Donald Knuth) и постпроцессора Томаса Рокики (Thomas Rokicki)

dvips . Все диаграммы компоновались автором с помощью утилиты pic2graph, управляющей

gpic , и grap2graph, управляющей

grap -реализацией Теда Фабера (Ted Faber) (утилита grap2graph была написана автором для данного проекта и в настоящее время является частью дистрибутива

groff). Вся инструментальная связка работала на стандартной системе Red Hat Linux.

Дизайн обложки — композиция двух изображений из оригинальных

Дзэн-комиксов (Zen Comics) Иоанны Саладжан (Ioanna Salajan). Он был адаптирован и раскрашен в основном Джерри Вотта (Jerry Votta) при участии автора.

Примечания

1

Три с половиной десятилетия между 1969 и 2003 гг. — это время исторической эволюции ОС Unix, воплотившей достижения более 50 млн. человеко-лет.

2

Оценка его работы со ссылками на Web-всрсии значимых частей находится на странице "Записки о Кристофере Александре" <http://www.math.utsa.edu/~salingar/Chris.text.html>.

3

Действительно, Ethernet уже дважды была заменена другой технологией с тем же названием: в первый раз, когда коаксиальный кабель был заменен витой парой, и во второй раз, когда появилась технология гигабитовой Ethernet.

4

С его именем неразрывно связаны такие понятия, как расстояние Хемминга (Hamming distance) и код Хемминга (Hamming code).

5

Джим Геттис, один из архитекторов системы X (и человек, внесший свой вклад в данную книгу), в статье

"The Two-Edged Sword" [29] глубоко размышляет о том, как стиль невмешательства X может продуктивно развиваться дальше. В этом эссе представлен ряд специфических рекомендаций и выражен характерный для Unix образ мышления.

6

Другие операционной системы в общем случае копируют или клонируют Unix-реализации TCP/IP. Их ошибка заключается в том, что они обычно не заимствуют имеющиеся в Unix прочные традиции экспертной оценки, примерами которых являются такие документы, как RFC 1025 (

TCP and IP Bake Off).

7

Первоначально это было сказано Стефаном С. Джонсоном (Stephen C. Johnson), который, вероятно, более известен как автор программы

yacc , об оснастке TSO в операционной системе IBM MVS.

8

Здесь представлено оригинальное дополнение Пайка (см. книгу Брукса, стр. 102). Ссылка указывает на раннее издание книги

"The Mythical Man-Month" [8]: "Покажите мне ваши блок-схемы, скройте таблицы, и я буду озадачен, покажите мне ваши таблицы и, скорее всего, блок-схемы мне не потребуются; они будут очевидны".

9

Джонатан Постел (Jonathan Postel) был первым редактором серии Internet-стандартов RFC и одним из главных архитекторов Internet. Памятная страница <http://www.postel.org/postel.html> поддерживается Центром Постела по экспериментальным сетям (Postel Center for Experimental Networking).

10

Полная цитата такова: "Нам следует забывать о небольшой эффективности, например, в 97% случаев: преждевременная оптимизация — корень всех зол". Сам Кнутт приписывал эту цитату Ч. Хоару (Charles Antony Richard Hoare). - Прм. авт.

11

Одним замечательным примером является статья Батлера Лампсона (Butler Lampson) "Рекомендации по проектированию компьютерных систем" (Hints for Computer System Design) [43], которые были обнаружены позднее в процессе подготовки данной книги. В статье не только выражены многие афоризмы Unix в формах, которые были открыты независимо, но и используется множество тех же ключевых фраз для их иллюстрации.

12

Кен Томпсон напомнил автору, что сегодняшние сотовые телефоны обладают большим объемом оперативной памяти, чем совокупный объем оперативной памяти и дискового пространства PDP-7.

Большой диск в то время имел емкость меньше одного мегабайта.

13

Существует Web-версия списка часто задаваемых вопросов (FAQ) по PDP-компьютерам <http://www.faqs.org/faqs/dec-faq/pdp8>, в котором по-другому трактуется роль компьютера PDP-7 в истории.

14

Руководства по седьмой версии доступны на странице http://plan9.bell-labs.com/7thEdMan/index.html.

15

UUCP считалась великолепным средством, когда

быстрый модем обеспечивал скорость 300 бод.

16

Серия PS/2, тем не менее, действительно оставила свой след в последующих компьютерах IBM PC — в этой серии мышь стала стандартным периферийным устройством. Именно поэтому разъем для подключения мыши на задней панели системного блока называется "PS/2-портом".

17

Игра "SPACEWAR" никак не была связана с игрой Кена Томпсона "Space Travel", кроме того, что обе они привлекали поклонников научно-технической фантастики.

18

Современные и исторические графики рыночных долей Web-серверов доступны в ежемесячном обзоре "Netcraft Web Server Survey" по адресу <http://www.netcraft.com/survey/>.

19

Для читателей, не имеющих опыта работы в Unix: канал представляет собой способ соединения вывода одной программы с вводом другой. Возможные варианты применения данной идеи для обеспечения взаимодействия программ рассматриваются в главе 7.

20

Периодические прерывания от аппаратного таймера (periodic clock interrupt) необходимы в виде тактовых импульсов для системы разделения времени. В каждом такте таймер сообщает системе о том, что можно переключиться на другую задачу, определяя длительность кванта времени. В настоящее время Unix-системы настроены на 60 или 100 тактовых импульсов в секунду.

21

Для обозначения этого подхода в настоящее время используется новомодное понятие

система безопасности на основе ролей (role-based security) .

22

Данная проблема весьма серьезно рассматривалась в корпорации Microsoft в ходе перестройки службы Hotmail. См. [7].

23

Более подробный анализ технических характеристик различных операционных систем приведен на Web-сайте OSData <http://www.osdata.com/>.

24

За исключением системы Multics, которая в основном оказывала влияние в период между опубликованием се спецификаций в 1965 году и ее фактической поставкой в 1969 году.

25

Подробности данного процесса описаны в статье Маршала Кирка Маккьюзика в [56].

26

Более подробная информация доступна на Web-сайте OpenVMS.org <http://www.openvms.org>

27

MacOS фактически состоит из двух частных уровней (перенесенные приложения OpenStep и классические GUI-интерфейсы Macintosh) поверх Unix-основы с открытым исходным кодом (проект Darwin).

28

Возвращаясь к вопросу о части технологии Amiga, IBM предоставила компании Commodore лицензию на свой язык сценариев REXX. Данная сделка описана на странице http://www.os2bbs.com/os2news/OS2Warp. html.

29

Адрес портала AmigaOS: <http://os.amiga.com/>.

30

Операционная система GEM <http://geocities.com/SiliconValley/Vista/6148/gem.htm>.

31

В качестве примера рекомендуются сайта OS Voice <http://www.os2voice.org/> и OS/2 BBS.COM <http://www.os2bbs.com/>.

32

Возможно. Это подтверждает тот факт, что унифицирующей метафорой для всех операционных систем Microsoft является тезис "потребитель должен быть замкнут".

33

http://security.tombom.co.uk/shatter.html

34

16 Корпорация Microsoft в марте 2003 года фактически публично признала, что NT-безопасность невозможна. См. Web-страницу http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS03-010.asp.

35

17 Проблема DLL hell в некоторой степени "смягчена" структурой разработки .NET, в которой осуществляется контроль версий, однако по состоянию на 2003 год технология .NET поставлялась только в профессиональные серверные версии Windows NT.

36

18 Технология Cygwin в значительной мере согласована с Единым стандартом Unix (Single Unix Specification), однако программы, требующие непосредственного доступа к аппаратным ресурсам, запускаются с ограничениями в ядре поддерживающей их Windows-системы. Так, широко известны проблемы с Ethernet-платами.

37

http://www.cbttape.org/cdrom.htm

38

Опытной машиной и первоначальной целью был компьютер серии 40 с измененным микрокодом, однако он оказался недостаточно мощным; действующая система была реализована на компьютере 360/67.

39

Результаты Linux-стратегии эмуляции и поглощения заметно отличаются от практики захвата и расширения, характерной для некоторых се конкурентов. Для начинающих: Linux не нарушает совместимости с тем, что эмулирует, и таким образом не привязывает клиентов к "расширенной" версии.

40

Закон Брукса гласит, что подключение к запаздывающему проекту новых программистов еще больше замедляет работу, В более широком смысле рост затрат и количества ошибок можно выразить квадратом числа, соответствующего количеству программистов, которые были задействованы в проекте.

41

Согласно модели Хаттона, небольшие различия в максимальном размере элемента кода, который программист может держать в краткосрочной памяти, непосредственно влияют на эффективность работы программиста.

42

В основном труде по рассматриваемой теме, а именно "

Refactoring" [21], автор фактически указывает, что принципиальная цель рефакторинга

состоит в усилении ортогональности. Однако ввиду недостаточно развитой концепции он может только аппроксимировать данную идею из необходимости устранения дублирования кода и других нежелательных явлений, многие из которых являются следствием нарушения ортогональности.

43

Типичным примером плохой организации кэширования является директива rehash в

csh(1) ; введите man 1 csh для получения более подробных сведений. Другой пример приведен в разделе 12.4.3.

44

Последний пример "переплетения" Unix и Дзэн приведен в приложении Г.

45

Архитектурный выбор порядка интерпретации битов в машинном слове —

"обратный" и

"прямой" (big-endian и

little-endian). Хотя канонический адрес отсутствует, поиск в Web по фразе

"On Holy Wars and a Plea for Peace" позволит найти классическую и интересную статью по данной теме.

46

Широко распространенное мнение о том, что автоинкрементная и автодекрементная функции вошли в C, поскольку они представляли машинные команды PDP-11, — это не более чем миф. Согласно воспоминаниям Денниса Ритчи, данные операции были предусмотрены в предшествующем языке B еще до появления PDP-11.

47

Наличие глобальных переменных также означает, что повторно использовать данный код

невозможно, то есть, множество экземпляров в одном процессе, вероятно, препятствуют работе друг друга.

48

Много лет назад я узнал из книги Кернигана и Плоджера

"The Elements of Programming Style" полезное правило: писать однострочный комментарий немедленно после прототипа функции, причем для вся без исключения функций.

49

Простейший способ сбора данных сведений заключается в анализе тег-файлов, сгенерированных с помощью таких утилит, как

etags(1) или

ctags(1) .

50

Существует легенда о том, что в некоторых ранних системах резервирования билетов на авиарейсы для учета количества пассажиров самолета выделялся ровно 1 байт. Очевидно, они были весьма озадачены появлением Boeing 747, первого самолета, который мог разместить на борту более 255 пассажиров.

51

Файлы паролей обычно считываются один раз в течение пользовательского сеанса во время регистрации в системе, а после этого иногда утилитами файловой системы, такими как

ls(1), которые должны преобразовывать числовые идентификаторы пользователей и групп в имена.

52

Не следует путать рассматриваемую здесь прозрачность конструкции с прозрачностью пикселей, которая поддерживается в PNG-изображениях.

53

Одним из пережитков этой предшествующей Unix истории является то, что Internet-протоколы обычно используют в качестве ограничителя строк последовательность CR-LF вместо принятой в Unix LF.

54

Документ RFC 3117: <ftp://ftp.frc-editor.org/in-notes/rfc3117.txt>.

55

Документ RFC 3205: <http://www.faqs.org/rfcs/rfc3205.html>.

56

См. RFC 2324 и RFC 2325 на страницах <http//www.ietf.org/rfc/rfc2324.txt> и <http//www.ietf.org/rfc/rfc2325.txt> соответственно.

57

Коллега, мыслящий экономическими категориями, комментирует: "Воспринимаемость относится к уменьшению входных барьеров, а прозрачность — к сокращению стоимости жизни в данном коде".

58

При разработке ядра Linux предпринимается множество попыток добиться воспринимаемости, включая подкаталог документации в архиве исходного кода ядра и значительное количество учебных пособий на Web-сайтах и в книгах. Скорость, с которой изменяется ядро, препятствует этим попыткам — документация хронически отстает.

59

Возвратное тестирование (regression testing) представляет собой метод для обнаружения ошибок, появляющихся по мерс модификации программного обеспечения. Оно состоит в периодической проверке вывода изменяющегося программного обеспечения для некоторого фиксированного тестового ввода по сравнению со снимком вывода, взятого на ранней стадии данного процесса. Известно (или предполагается) что данный снимок корректен.

60

Фактически переменная TERM устанавливается системой во время регистрации пользователя. Для реальных терминалов на последовательных линиях преобразование из tty-строк в значения TERM, устанавливается из системного конфигурационного файла во время загрузки системы; детали весьма различаются в разных Unix-системах. Эмуляторы терминалов, такие как

xterm(1) , устанавливают данную переменную самостоятельно.

61

Ранним предком таких проверочных программ в операционной системе Unix была утилита

lint , программа проверки C-кода, обособленная от компилятора C. Хотя GCC включает в себя ее функции, приверженцы старой школы Unix до сих пор склонны называть процесс запуска такой программы "линтингом" (linting), а ее имя сохранилось в названии таких утилит, как

xmllint .

62

Инвариантным является свойство конструкции программного обеспечения, которое сохраняется при каждой операции в нем. Например, в большинстве баз данных инвариантным свойством является то, что никакие две записи не могут иметь один и тот же ключ. В C-программе, корректно обрабатывающей строки, каждый строковый буфер должен содержать завершающий NUL-байт на выходе из каждой строковой функции. В системах инвентаризации пи один счетчик частей не может быть отрицательным.

63

См. результаты, приведенные в

"Improving Context Switching Performance of Idle Tasks under Linux" [1].

64

setuid-программа выполняется не с привилегиями вызвавшего ее пользователя, а с привилегиями владельца исполняемого файла. Эту особенность можно использовать для предоставления ограниченного, программно-управляемого доступа к таким элементам, как файлы паролей, непосредственное изменение которых должно быть разрешено только администраторам.

65

То есть, лучшие достижения, выраженные в количестве прорывов безопасности относительно общего времени работы в Internet.

66

Распространенная ошибка в программировании с созданием подоболочек заключается в том, что программист забывает блокировать сигналы в родительском процессе при работающем подпроцессе. Без такой предосторожности прерывание подпроцесса может иметь нежелательные побочные эффекты для родительского процесса.

67

По сути, это излишнее упрощение. Дополнительная информация представлена при обсуждении переменных EDITOR и VISUAL в главе 10.

68

В справочной странице

less(1) сказано, что

less противоположна

more .

69

Распространенная ошибка заключается в использовании вместо “$@” выражения $*. Это приводит к негативным последствиям при передаче имени файла, содержащего пробелы.

70

Стандартный ввод и стандартный вывод программы

qmail-popup представляют собой сокеты, а стандартная ошибка (дескриптор файла 2) отправляется в log-файл. Гарантируется, что дескриптор файла 3 будет следующим выделенным дескриптором. В одном печально известном комментарии к ядру было сказано: "Никто и не ждет, что вы это поймете".

71

Коллега, порекомендовавший этот учебный пример, прокомментировал его так: "Да, можно выйти из положения с помощью этой методики..., если имеется всего несколько легко распознаваемых блоков информации, поступающих обратно от подчиненного процесса, а также есть щипцы и противорадиационный костюм".

72

Особенно опасным вариантом этой атаки является вход

в именованный Unix-сокет, где программы, создающие и использующие данные, пытаются найти временный файл.

73

"Конкуренция" (race condition) представляет собой класс проблем, в которых корректное поведение системы зависит от двух независимых событий, происходящих в правильном порядке, однако отсутствует механизм, для того чтобы гарантировать фактическое возникновение этих событий. Конкуренция приводит к появлению периодических проблем с временной зависимостью, которые могут создавать значительные трудности при отладке.

74

Средство STREAMS было гораздо более сложным. Деннису Ритчи приписывают следующее высказывание: "Слово "streams" означает нечто другое, если его прокричать".

75

Разработчики основного конкурирующего с GNOME пакета KDE вначале использовали технологию CORBA, но отказались от нес в версии 2.0. С тех пор они ищут более легковесные IPC-методы.

76

Лес Хаттон в письме, где речь идет о его очередной книге,

Software Failure , сообщает: "Если для измерения плотности дефектов учитывать только выполняемые строки, то плотность дефектов в зависимости от языка почти на порядок меньше плотности дефектов в зависимости от квалификации инженера".

77

Для менее технически подготовленных читателей: скомпилированная форма С-программы производится из ее исходного С-кода путем компиляции и связывания. PostScript-версия troffрументов troff-документа является производной от исходного troff-кода; чтобы осуществить это преобразование используется команда

troff . Существует множество других видов производных. Почти все они могут быть выражены с помощью make-файлов.

78

Любой язык Тьюринга мог бы теоретически использоваться для универсального программирования и теоретически является в точности таким же мощным, как любой другой язык Тьюринга. На практике некоторые языки Тьюринга были бы слишком сложными для использования за пределами специфической или узкой предметной области.

79

Стандарт POSIX для регулярных выражений вводит некоторые символьные диапазоны, такие как [[:lower:]] и [[:digit:]]. Кроме того, отдельные специфические средства используют

дополнительные символы-шаблоны, не описанные здесь. Однако для интерпретации большинства регулярных выражений приведенных примеров достаточно.

80

Для тех, кто не является Unix-программистом: инструментарий X представляет собой графическую библиотеку, которая предоставляет GUI-объекты (такие как надписи, кнопки и выпадающие меню) подключенным к ней программам. В большинстве других операционных систем с графическим интерфейсом предоставляется только один инструментарий, используемый всеми. Unix и сервер X поддерживают несколько инструментариев. Это является частью разделения политики и механизма, которое в главе 1 было названо целью проектирования X-сервера. GTK и Qt являются двумя наиболее популярными X-инструментариями с открытым исходным кодом.

81

Однако неочевидно, что XSLT мог бы быть несколько проще при тех же функциональных возможностях, поэтому его нельзя охарактеризовать как плохую конструкцию.

82

Концепции и практическое применение XSL
<http://nwalsh.com/docs/tutorials/xsl/xsl/slides.html>.

83

http://www.netlib.org/

84

Включать собственные иллюстрации как примеры кода также весьма традиционно для книг по Unix, в которых описывается программа

pic(1) .

85

Цитата принадлежит Алану Перлису (Alan Perils), который провел ряд передовых работ по модульности программного обеспечения приблизительно в 1970 году. Двоеточие в данном случае означало разделитель или ограничитель операторов в различных потомках языка Algol, включая Pascal и C.

86

Для читателей, никогда не программировавших на современных языках сценариев: словарь представляет собой таблицу поиска связей ключ-значение, часто реализуемую посредством хэш-таблицы. C-программисты тратят большую часть времени кодирования, реализуя словари различными сложными способами.

87

Когда-то я сам был мастером

awk , но кто-то напомнил мне, что данный язык применим к проблеме создания HTML-документов, поэтому единственный awk-пример в данной книге связан именно с ней.

88

Существует сайт проекта GhostScript <http://www.cs.wise.edu/~ghost/>.

89

Первое руководство по PostScript <http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html>.

90

Реализации JavaScript с открытыми исходными кодами на C и Java доступны на сайте <http://www.mozilla.org/js/>.

91

16 20 миллионов — сдержанная оценка, основанная на графиках Linux Counter и других источниках по состоянию на середину 2003 года.

92

17 Программа

Kmail , которая рассматривалась в главе 6, по этой причине даже не отслеживает внешние ссылки в HTML-документах.

93

Дальнейшее развитие этой точки зрения приведено в книге [3].

94

Языки сценариев часто решают данную проблему более изящно, чем C. Для того чтобы понять, как именно они это делают, следует изучить методику

потоковых документов (here documents) в shell и конструкцию тройных кавычек в Python.

95

Здесь CGI означает не Computer Graphic Imagery, а технологию Common Gateway Interface (интерфейс общего шлюза), которая применяется для создания интерактивных Web-документов.

96

Для отображения скрытых файлов используется параметр -a утилиты

ls(1) .

97

Суффикс "rc" связан с системой, предшествующей Unix, CTSS. В ней присутствовала функция сценария команд, которая называлась "runcom". В ранних Unix-системах имя "rc" использовалось для загрузочного сценария операционной системы как дань runcom в CTSS.

98

Никто не знает действительно изящного способа представить эти распределенные данные о настройках. Переменные среды, вероятно, не являются этим способом, однако для всех известных альтернатив характерны одинаково неприятные проблемы.

99

4 В действительности, большинство Unix-программ вначале проверяют переменную VISUAL, и только если она не установлена, обращаются к переменной EDITOR. Это пережиток того времени, когда пользователи имели различные настройки для строковых и визуальных редакторов.

100

См. стандарты GNU-программирования на странице <http://www.gnu.org/prep/standards.html>.

101

Файл .xinitrc является аналогом каталога автозагрузки в Windows и других операционных системах.

102

Диспетчер окон (window manager) поддерживает связи между окнами на экране и запущенными заданиями. Диспетчер окон управляет такими функциями, как расположение, отображение заголовков, свертывание, развертывание, перемещение, изменение размеров и затенение окон.

103

Такой взгляд на проблему характерен для нетехнического конечного пользователя.

104

Исходные коды данной программы и других конвертеров с подобными интерфейсами доступны на странице <http://ww.cdrom.com/pub/png>.

105

Управляющая программа (harness programm) представляет собой упаковщик, предназначенный для того, чтобы предоставлять вызываемым им программам доступ к некоторому специальному ресурсу. Данный термин часто употребляется при характеристике средств тестирования, которые предоставляют тестовые нагрузки и (часто) примеры корректного вывода для проверки фактического вывода тестируемых программ.

106

СтраницаScript-Fu <http://www.xcf.Berkeley.edu/~gimp/script-fu/script-fu.html>.

107

Если в системе поддерживаются подсвечиваемые всплывающие окна, которые "мало вторгаются" между пользователем и приложением,

используйте их .

108

Для читателей, не знакомых с O-нотацией: она представляет собой способ указания зависимости между средним временем выполнения алгоритма и размерами его входных данных. Алгоритм O(1) выполняется за постоянное время. Алгоритм O(

n) выполняется за время, которое можно вычислить по формуле: An + C, где A — некоторый неизвестный постоянный коэффициент пропорциональности, а C — неизвестная константа, представляющая время установки. Линейный поиск определенного значения в списке

представляет собой алгоритм типа O(

n). Алгоритм O(

n ?) выполняется за время

An ? плюс величина более низкого порядка (которая может быть линейной либо логарифмической или любой другой функцией ниже квадратичной). Поиск повторяющихся значений в списке (примитивным методом, без сортировки списка) является алгоритмом O(

n ?). Аналогично, алгоритмы порядка O(

n ?) имеют среднее время выполнения, вычисляемое по кубической формуле. Такие алгоритмы часто слишком медленны для практического применения. Порядок O(

log n) типичен для поиска по дереву. Взвешенный выбор алгоритма часто может сократить время выполнения с O(

n ?) до O(

log n). Иногда, когда требуется рассчитать использование алгоритмом памяти, можно заметить, что оно изменяется как O(1) или O(

n), или O(

n ?). Как правило, алгоритмы с использованием памяти O(

n ?) или более высокого порядка являются непрактичными.

109

Удвоение мощности в течение каждых 18 месяцев, обычно цитируемое в контексте закона Мура, подразумевает, что можно достичь 26% прироста производительности просто путем приобретения нового аппаратного обеспечения через 6 месяцев.

110

Термины, введенные здесь для обозначения проблем проектирования, происходят из устоявшегося жаргона хакеров, описанного в книге [66].

111

Разделение случайной и необязательной сложности означает, что рассматриваемые здесь категории

не являются тем же, что сущность и случайность в очерке Фреда Брукса

"No Silver Bullet" [8], однако в философском смысле они имеют одинаковое происхождение.

112

Молодые читатели могут не знать, что раньше терминалы печатали (на бумаге и очень медленно).

113

http://plan9.bell-labs.com/sys/doc/sam/sam.html

114

Разработчиками Emacs были Ричард М. Столлмен (Richard M. Stallman) и Берни Гринберг (Bernie Greenberg). Первоначальный редактор Emacs был изобретением Столлмена, первая версия со встроенным языком Lisp была создана Гринбергом, а окончательная на сегодняшний день версия создана Столлменом на основе версии Гринберга. К 2003 году нет полного изложения истории разработки редактора, но эту тему освещает статья Гринберга

"Multics Emacs: The History, Design, and Implementation", которую можно найти в Web по ключевым словам.

115

http://www.cs.yorku.ca/~oz/wily

116

http://plan9.bell-labs.com/sys/doc/acme/acme.html

117

Подробности на странице <ftp://ftp.idiom.com/pub/compilers-list/free-compilers> (Список бесплатных компиляторов и интерпретаторов).

118

За пределами мира Unix этот прирост аппаратной производительности на три порядка в значительной мере затмевается соответствующим понижением производительности программ.

119

Серьезность данной проблемы подтверждается богатым сленгом, выработанным Unix-программистами для описания различных ее разновидностей: "псевдонимная ошибка" (aliasing bug), "нарушение выделенной области памяти" (arena corruption), "утечка памяти" (memory leak), "переполнение буфера" (buffer overflow), "разрушение стека" (stack smash), "отклонение указателя" (fandango on core), "недействительный указатель" (stale pointer), "подкачка памяти" (heap trashing), а также вызывающее справедливые опасения "вторичное повреждение" (secondary damage). Пояснения приведены в Словаре хакера <http://www.catb.org/~esr/jargon>.

120

Последний стандарт C++, датированный 1998 годом, был широко распространенным, но слабым, особенно в области библиотек.

121

См. очерк Тома Кристиансена (Tom Christiansen)

"Csh Programming Considered Harmful" который можно легко найти в Web.

122

Существует Web-сайт проекта Freenet <http://freenetproject.org>.

123

URL-запроса для получения статистических данных:

<http://sourceforge.net/softwaremap/trove_list.php?form_cat=160>.

124

URL-запроса для получения статистических данных:
<http://freshmeat.net/browse/160/?topic_id=160>.

125

Распространенное более новое описание языка Perl как "швейцарской бензопилы" является производным.

126

PLY можно загрузить со страницы <http://systems.cs.uchicago.edu/ply/>.

127

Пакет CUP доступен на странице <http://www.cs.princeton.edu/~appel/modern/java/CUP>.

128

Пакет Jack доступен на странице
<http://www.javaworld.com/javaworld/jw-12-1996/jw-12-jack.html>.

129

Пакет Yacc/M доступен на странице <http://david.tribble.com/yaccm.html>.

130

http://cm.bell-labs.com/cm/cs/upe/

131

Статья доступна в Web: <http://www.tip.net.au/~millerp/rmch/recu-make-cons-harm.html>.

132

Блочный тест представляет собой тестовый код, прикрепленный к модулю для проверки корректности представления. Использование термина "блочный тест" подразумевает, что тест написан разработчиком одновременно с основным кодом и означает порядок, при котором версии модуля не считаются завершенными до тех нор, пока к ним не прикреплен тестовый код. Данный термин и идея возникли в методологии "Экстремального программирования", популяризированной Кентом Беком (Kent Beck), но получили широкое распространение среди Unix-программистов примерно с 2001 года.

133

Более подробная информация по данным и родственным командам управления компиляцией приведена в справочном меню Emacs: p+processes->compile.

134

Подробнее эти и родственные команды описываются в подразделе справочной системы Emacs, озаглавленном

Version Control (Управление версиями).

135

Агентство NASA, в котором целенаправленно создавалось программное обеспечение, предназначенное для использования в течение десятилетий, научилось настаивать на доступности исходного кода для всех создателей программного обеспечения для аэрокосмической техники.

136

Обе системы PDP-7 Unix и Linux стали примерами "неожиданной живучести". Unix в качестве "игрушки для опытов" была создана несколькими исследователями в перерывах между основными проектами. Наполовину она была предназначена для экспериментов с файловой системой и наполовину для поддержки компьютерной игры. Linux была охарактеризована своим создателем как "мой эмулятор терминала с выросшими ногами" [85]

137

Таким образом, литера "C" в названии языка C означает Common (общий) или, возможно, "Christopher". Аббревиатура BCPL первоначально расшифровывалась как "Bootstrap CPL" — сильно упрощенная версия языка CPL, весьма интересного, но слишком претенциозного языка общего программирования (Common Programming Language) Оксфордского и Кембриджского университетов, также называемого "языком программирования Кристофера" (Christopher Programming Language) по имени его главного пропагандиста, первопроходца компьютерной науки Кристофера Стрэчи (Christopher Strachey).

138

Документ доступен в Web: <http://anubis.dkuug.dk/JTC1/SC22/WG14/www/charter>.

139

Первоначальный пробный стандарт в 1985 году назывался IEEE-IX. Название "POSIX" было предложено Ричардом Столлменом. Введение в POSIX.1 гласит "Ожидается произношение "поз-икс" как "позитив", а не "по-сикс" или в других вариантах. Произношение опубликовано в целях обнародования стандартного способа ссылки на стандартный интерфейс операционной системы".

140

Один Linux-дистрибьютор, а именно Лазермун (Lasermoon) из Великобритании, добился сертификации POSIX.1 FIPS 151-2, но вышел из бизнеса, поскольку потенциальных клиентов сертификация не интересовала.

141

Эта тема обсуждается в книге

"Just for Fun " [85]

142

Web-поиск, вероятно, предоставит популярную страницу, на которой сатирически описывается семиуровневая модель OSI сравнивается.

143

Эти слова впервые были произнесены членом руководства IETF Дэйвом Кларком (Dave Clark) в 1992 году на бурном собрании, в ходе которого IETF отвергла протокол открытого взаимодействия систем.

144

9 RFC 1149 доступен в Web — <http://www.ietf.org/rfc/rfc1149.txt>. Его реализация описана на стр. <http://www.blug.linux.no/rfc1149/writeup.html>.

145

10 RFC 2324 доступен в Web — <http://www.ietf.org/rfc/rfc2324.txt>.

146

11 RFC 3514 доступен в Web — <http://www.ietf.org/rfc/rfc3514.txt>.

147

В XML-жаргоне то, что здесь названо "диалектом" (dialect), называется "приложением" (application). Автор избегает использования слова "приложение", поскольку в данном случае оно противоречит другому более широко распространенному его значению.

148

В течение нескольких лет казалось, что семиуровневый стандарт ISO может успешно конкурировать с набором протоколов TCP/IP. Он продвигался Европейским комитетом стандартов, напуганным мыслью о заимствовании любой технологии, рожденной в недрах Пентагона. Увы, их негодование превысило остроту их технического зрения. Результат оказался чрезмерно сложным и напрасным. Более подробно эта тема описана в книге [60].

149

Это название — дань кинофильму, вышедшему в 1958 году, который вошел в историю как "наихудшее из созданного",

"Plan 9 from Outer Space". Документацию, включая обзорную статью, описывающую архитектуру, наряду с полным исходным кодом и дистрибутивом, который инсталлируется на PC, можно без труда найти с помощью Web-поиска по фразе "Plan 9 from Bell Labs".

150

История о том, как была создана UTF-8, включает в себя описание безумной ночной работы Кена Томпсона и Роба Пайка — <http://www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt>.

151

Ищите F_NOTIFY в

fcntl(2) .

152

Данный параграф основывается на аналитической статье Генри Спенсера, вышедшей в 1984 году. Он отметил, что управление задачами было необходимо и целесообразно точно учесть в POSIX.1 и последующих стандартах Unix,

поскольку оно "просачивается" в каждую программу и, следовательно, должно быть продумано в любом интерфейсе "приложение-система". Отсюда и одобрение POSIX ошибочной конструкции, когда правильные решения "выходили за рамки", а следовательно, даже не рассматривались.

153

Web-страница проекта

screen(1) — http://www.math.fu-berlin.de/~guckes/screen/.

154

Для непрограммистов:

обработка исключительных ситуаций — способ, с помощью которого программа прерывается в середине процедуры. Это не совсем то же, что выход, поскольку такой останов может быть обработан кодом ловушки во включающей его процедуре. Исключительные ситуации обычно используются для сигнализации об ошибках или неожиданных обстоятельствах, которые означают, что продолжение обычной работы нецелесообразно.

155

http://www.cros-os.org/

156

Что же касается операционной системы Apple Newton, мини-компьютера AS/400 и карманного компьютера Palm, то здесь речь может идти об исключении.

157

Более полное обсуждение данного эффекта приведено в главе

"The Magic Cauldron" книги [67].

158

Весьма пугающий перечень возможностей, созданный известным специалистом по безопасности, приведен в TCPA FAQ <http://www.cl.cam.ac.uk/~rja14/tcpa-faq.html>.

159

Введение в гибкое программирование представлено на странице Agile Manifesto <http://agilemanifesto.org/>.

160

1 Книга

"The Tao of Programming" доступна в Web — <http://www.canonical.org/~kragen/tao-of-programming.html>.

161

2 Книга

"AI Koans" доступна в Web <http://www.catb.org/~esr/jargon/html/Some-AI-Koans.html>.

162

Книга

"Loginataka" доступна в Web <http://www.catb.org/~esr/faqs/loginataka.html>.

163

Книга

"Tales of Zen Master Greg" доступна в Web <http://www.gu.uwa.edu.au/users/greg/>.

164

Книга

"Gateless Gate" доступна в Web <http://www.ibiblio.org/zen/cgi-bin/koan-index.pl>.