ОГЛАВЛЕНИЕ

13
ЦЕЛОЕ И ЧАСТИ

Я духов вызывать могу из бездны.
И я могу, и каждый может.
Вопрос лишь, явятся ль на зов они?*

Шекспир. Король Генрих IV

* Перевод Е. Бируковой. — *Примеч. ред.*

Среди современных волшебников, как и встарь, встречаются хвастуны: «Я могу писать программы, которые управляют воздушным движением, перехватывают баллистические ракеты, сверяют банковские счета, контролируют производственные линии». На что приходит ответ: «Я тоже могу, и любой человек может, но будут ли они работать, когда вы их напишете?»

Как написать работающую программу? Как ее протестировать? И как интегрировать протестированный набор компонентных программ в протестированную и надежную систему? Мы уже касались этих методов здесь и там; давайте теперь рассмотрим их несколько системно.

ДИЗАЙН, ИСКЛЮЧАЮЩИЙ ОШИБКИ

ОБЕСПЕЧЕНИЕ ДЕФЕКТОСТОЙКОСТИ ОПРЕДЕЛЕНИЯ. Наиболее пагубными и едва уловимыми дефектами являются системные ошибки, возникающие из-за несовпадающих допущений, принимаемых авторами разных компонентов. Подход к концептуальной целостности, рассмотренный выше в главах 4, 5 и 6, затрагивает эти проблемы непосредственно. Одним словом, концептуальная целостность продукта не только упрощает его использование, но и облегчает его разработку и делает менее подверженным ошибкам.

Такую же работу выполняют детальные, кропотливые архитектурные усилия, вытекающие из указанного подхода. В. А. Высоцкий из проекта Safeguard, выполнявшегося в *Bell Telephone Laboratories*, говорит, что «важнейшая работа — дать продукту определение. Многие, очень многие неудачи касаются именно тех аспектов, которые никогда не были точно определены».[1] Тщательное определение функций, тщательная спецификация и дисциплинированное изгнание излишеств функциональности и полетов технической

мысли — все это уменьшает число системных ошибок, которые должны быть найдены.

ТЕСТИРОВАНИЕ СПЕЦИФИКАЦИИ. Задолго до того, как появится какой-либо код, спецификация должна быть передана группе внешнего тестирования для тщательного изучения на предмет полноты и ясности. Как говорит Высоцкий, сами разработчики не могут этого сделать: «Они не могут признаться, что не понимают ее, они будут счастливо прокладывать свой путь через пропущенные и темные места».

НИСХОДЯЩИЙ ДИЗАЙН. В подробной статье 1971 года Никлаус Вирт (Niklaus Wirth) формализовал процедуру дизайна, которая в течение многих лет использовалась лучшими программистами.[2] Более того, его понятия, хотя и сформулированные для дизайна программ, полностью применимы к дизайну комплексных систем программ. Деление процесса сборки системы на архитектуру, имплементацию и реализацию является воплощением этих понятий; более того, каждая архитектура, имплементация и реализация могут быть наилучшим образом выполняться нисходящими методами.

Вкратце, процедура Вирта состоит в том, чтобы определить дизайн как последовательность *уточняющих шагов*. Делается грубый набросок определения работы и грубый метод ее решения, который достигает принципиального результата. Затем указанное определение изучается более внимательно с целью увидеть, как результат отличается от того, что требуется, а крупные шаги решения разбиваются на более мелкие. Каждое уточнение в определении задачи становится уточнением в алгоритме решения, и каждое может сопровождаться уточнением в представлении данных.

Из этого процесса выявляют *модули* решения или модули данных, дальнейшее уточнение которых может происходить независимо от другой работы. Степень этой модульности определяет адаптивность и изменчивость программы.

Вирт выступает за использование как можно более высокоуровневой нотации на каждом этапе, раскрывая концепции и скрывая детали до тех пор, пока не станет необходимым дальнейшее уточнение.

Хороший нисходящий дизайн позволяет избежать ошибок несколькими путями. Во-первых, ясность структуры и представления облегчает точное изложение требований и функций модулей. Во-вторых, разделение и независимость модулей позволяют избежать системных ошибок. В-третьих, подавление деталей делает недостатки в структуре более очевидными. В-четвертых, дизайн может быть протестирован на каждом шаге его уточнения, поэтому тестирование можно начать раньше и сосредоточиться на надлежащем уровне детализации на каждом шаге.

Процесс поэтапного уточнения не означает, что никогда не нужно возвращаться назад, отбрасывать верхний уровень и начинать все сначала, когда он сталкивается с какой-то неожиданно затруднительной деталью. Действительно, такое случается часто. Но при этом гораздо легче понять, когда и почему нужно отбросить проект и начать все сначала. Многие плохие системы возникают из попытки спасти плохой базовый дизайн и залатать косметическими заплатками. Нисходящий дизайн уменьшает такой соблазн.

Убежден, что нисходящий дизайн является самой важной новой формализацией программирования последнего десятилетия.

СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ. Еще один важный набор новых идей для бездефектного дизайна в программах во многом вытекает из работ Дейкстры[3] и построен на теоретической структуре Бёма (Boehm) и Якопини (Jacopini).[4]

По сути, указанный подход заключается в дизайне программ, управляющие структуры которых состоят только из циклов, заданных инструкцией языка, такой как `DO WHILE`, и условных частей, разделенных на группы инструкций, помеченных скобками и обус-

ловленных конструкцией `IF ... THEN... ELSE`. Бом и Якопини показывают, что эти структуры являются теоретически достаточными; Дейкстра утверждает, что альтернативное, неограниченное ветвление через `GO TO` производит структуры, которые подвержены логическим ошибкам.

В основе, несомненно, лежат здравые мысли. Было высказано много критических замечаний, и дополнительные управляющие структуры, такие как множественное ветвление (так называемая инструкция `CASE`), для различения среди непредвиденных обстоятельствах и урегулирования аварийных ситуаций (`GO TO ABNORMAL END`) оказались очень удобными. Более того, некоторые критики заняли очень доктринерскую позицию в отношении того, чтобы избегать всех переходов `GOTO`, и это кажется уже чрезмерным.

Значимым моментом и жизненно важным для создания бездефектных программ является то, что мы хотим думать об управляющих структурах системы как об управляющих структурах, а не как об отдельных инструкциях ветвления. Такой образ мышления является важным шагом вперед.

ОТЛАДКА КОМПОНЕНТОВ

За последние 20 лет процедуры отладки программ для устранения ошибок прошли через большой цикл, и в некотором смысле они вернулись к тому, с чего начинали. Указанный цикл прошел четыре шага, и любопытно проследить их и увидеть мотивацию для каждого.

ОТЛАДКА НА МЕСТЕ. Ранние машины имели относительно плохое оборудование ввода-вывода и длинные задержки ввода-вывода. Как правило, машина считывала и писала на бумажную или магнитную ленту, и автономные средства обеспечения исполь-

зовались для подготовки ленты и печати. Это делало ленточный ввод-вывод невыносимо неудобным для отладки, поэтому вместо него использовалась консоль. Таким образом, отладка организовывалась так, чтобы обеспечить как можно больше тестов за машинный сеанс.

Программист составлял тщательный дизайн своей процедуры отладки, планируя, где остановиться, какие области памяти проверять, что там найти и что делать, если он не найдет. Это дотошное программирование отладочной логики само по себе могло занимать половину времени от написания отлаживаемой программы.

Смертный грех состоял в том, чтобы смело нажимать кнопку START, не разбив программу на тестовые секции с запланированными остановами.

ДАМПЫ ПАМЯТИ. Отладка на месте была очень эффективной. За двухчасовой сеанс можно было сделать около дюжины снимков. Но компьютеры были очень редки и очень дороги, и мысль о том, что все это машинное время будет потрачено впустую, была ужасающей.

Поэтому когда высокоскоростные принтеры были подключены к сети, методика изменилась. Теперь программа выполнялась до тех пор, пока проверка не нарушалась, а затем делался дамп всей памяти. Потом начиналась кропотливая работа за письменным столом, в которой изучалось содержимое каждой ячейки памяти. Время за письменным столом не сильно отличалось от времени отладки на месте; но оно происходило после выполнения теста, при расшифровке, а не до, при планировании. У любого конкретного пользователя отладка занимала гораздо больше времени, поскольку тестовые снимки зависели от пакетного оборотного времени. Вся процедура, однако, была составлена так, чтобы минимизировать потребление компьютерного времени и обслуживать как можно больше программистов.

СНИМКИ. Машины, на которых было разработано дампирование (выгрузка дампа) памяти, имели память емкостью 2000–4000 слов, или 8–16 Кбайт. Но размеры памяти росли семимильными шагами, и полная выгрузка дампов стала непрактичной. Поэтому люди разработали методы выборочных дампов, выборочной трассировки и вставки снимков в программы. TESTRAN в OS/360 является вершиной в линейке подобных средств, позволяя вставлять снимки в программу без повторной сборки или перекомпиляции.

ИНТЕРАКТИВНАЯ ОТЛАДКА. В 1959 году Кодд (Codd) и его коллеги[5] и Стрейчи[6], каждый в отдельности, сообщили о работе, целью которой была отладка в режиме разделения времени, позволяющая одновременно достичь мгновенной оборачиваемости отладки в активном режиме и эффективно использовать машинное время, как при пакетной обработке заданий. Компьютер имел несколько программ в памяти, готовых к исполнению. Терминал, управляемый только программой, ассоциировался с каждой отлаживаемой программой. Отладка осуществлялась под контролем управляющей программы. Когда программист на терминале останавливал свою программу с целью проверки хода выполнения или внесения изменений, супервизор запускал еще одну программу, таким образом оставляя машины все время занятыми.

У Кодда была разработана система мультипрограммирования, но акцент был сделан на повышение пропускной способности за счет эффективного использования ввода-вывода, а интерактивная отладка не была имплементирована. Идеи Стрейчи были усовершенствованы и имплементированы в 1963 году Корбато и его коллегами из MIT[7] в экспериментальной системе для 7090. Эта разработка привела к появлению MULTICS, TSS и других современных систем с совместным использованием времени.

Главными ощущаемыми пользователем различиями между отладкой в активном режиме, как она осуществлялась ранее, и сегодняшней интерактивной отладкой являются возможности, полученные

в результате присутствия программы-супервизора и связанных с ней интерпретаторов языков программирования. Можно программировать и производить отладку на языках высокого уровня. Эффективные средства редактирования позволяют легко делать изменения и моментальные снимки.

Возврат к способности мгновенного оборота отладки на месте еще не привел к возврату к предварительному планированию сеансов отладки. В каком-то смысле такое предпланирование уже не является таким необходимым, как раньше, поскольку машинное время не пропадает даром, пока человек сидит и думает.

Тем не менее интересные экспериментальные результаты Голда (Gold) показывают, что в три раза больше прогресса в интерактивной отладке достигается при первом взаимодействии каждого сеанса, чем при последующих взаимодействиях.[8] Это убедительно свидетельствует о том, что мы не реализуем потенциал взаимодействия из-за отсутствия планирования сессий. Пришло время стряхнуть пыль со старой методики отладки.

Считаю, что надлежащее использование хорошей терминальной системы требует двух часов за письменным столом для каждого двухчасового сеанса за терминалом. Половина этого времени уходит на уборку после последнего сеанса: обновление журнала отладки, заполнение обновленных списков программ в системной записной книжке, объяснение странных явлений. Другая половина уходит на подготовку: планирование изменений, улучшений и разработку детальных тестов для следующего раза. Без такого планирования трудно оставаться продуктивным в течение двух часов. Без постсессионной уборки трудно поддерживать порядок последующих терминальных сессий систематическим и поступательным.

ТЕСТОВЫЕ СЛУЧАИ. Что касается разработки фактических отладочных процедур и тестовых случаев, то Груенбергер (Gruenberger) имеет

особенно хорошую трактовку[9], и есть более короткие трактовки в других стандартных текстах.[10,11]

ОТЛАДКА СИСТЕМЫ

Неожиданно трудной частью разработки системы программирования является системный тест. Я уже говорил о некоторых причинах, вызывающих как трудности, так и неожиданности в этом процессе. Исходя из всего этого можно быть уверенным в двух вещах: отладка системы займет больше времени, чем можно ожидать, и ее трудность оправдывает тщательно систематизированный и плановый подход. Давайте теперь посмотрим, что включает в себя такой подход.[12]

ЗАДЕЙСТВОВАТЬ ОТЛАЖЕННЫЕ КОМПОНЕНТЫ. Здравый смысл, если не обычная практика, говорит нам, что отладку системы следует начинать только после того, как ее части, по-видимому, работают.

Обычная практика отходит от этого двумя путями. Во-первых, это подход «прикрутить все вместе и попробовать». Указанный подход, похоже, основан на том, что в дополнение к дефектам компонентов будут существовать системные (то есть интерфейсные) ошибки. Чем скорее вы сложите части вместе, тем скорее проявятся системные ошибки. Несколько менее сложным является представление о том, что, используя части для тестирования друг друга, можно избежать большого количества тестовых строительных лесов. Оба этих утверждения являются очевидными, но опыт показывает, что они не являются полной правдой — использование чистых, отлаженных компонентов экономит гораздо больше времени при тестировании системы, чем возведение строительных лесов и тщательное тестирование компонентов.

Немного более тонким является подход *документированной ошибки*. Он говорит о том, что компонент готов войти в тест системы,