БИБЛИОТЕКА ПРОГРАММИСТА

Э. Хант, Д. Томас

Программист-прагматик

Путь от подмастерья к мастеру

- Как бороться с недостатками программного обеспечения
- Как создать динамичную и адаптируемую программу
- Как осуществлять эффективное тестирование
- Как формировать команды программистов-прагматиков

Addison Wesley

ПИТЕР

Annotation

Находясь на переднем крае программирования, книга "Программист-прагматик. Путь от подмастерья к мастеру" абстрагируется от всевозрастающей специализации и технических тонкостей разработки программ на современном уровне, чтобы исследовать суть процесса – требования к работоспособной и поддерживаемой программе, приводящей пользователей в восторг. Книга охватывает различные темы – от личной ответственности и карьерного роста до архитектурных методик, придающих программам гибкость и простоту в адаптации и повторном использовании.

Прочитав эту книгу, вы научитесь:

Бороться с недостатками программного обеспечения;

Избегать ловушек, связанных с дублированием знания;

Создавать гибкие, динамичные и адаптируемые программы;

Избегать программирования в расчете на совпадение;

Защищать вашу программу при помощи контрактов, утверждений и исключений;

Собирать реальные требования;

Осуществлять безжалостное и эффективное тестирование;

Приводить в восторг ваших пользователей;

Формировать команды из программистов-прагматиков и с помощью автоматизации делать ваши разработки более точными.

- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54

- 55
- 56

Эндрю Хант, Дэвид Томас
Программист-прагматик
Путь от подмастерья к мастеру

Высказывания программистов-практиков о книге "Программист-прагматик"

Главное в этой книге то, что она поддерживает процесс создания программ в хорошей форме. [Книга] способствует вашему постоянному росту и явно написана людьми, знающими толк в программировании.

Кент Бек, автор книги Extreme Programming Explained: Embrace Change

Я обнаружил, что эта книга является смесью убедительных советов и замечательных аналогий!

Мартин Фаулер, автор книг Refactoring и UML Distilled

Я бы купил книгу, прочел ее дважды, а затем сказал бы всем моим коллегам, чтобы они скорее бежали в магазин и купили себе по экземпляру. Эту книгу я никогда не дал бы никому почитать, так как сходил бы с ума от беспокойства за ее сохранность.

Кевин Руланд, сотрудник отдела менеджмента фирмы MSG-Logistics

Мудрость и практический опыт авторов очевидны. Разделы, представленные в книге, уместны и полезны… Сильнейшее впечатление на меня произвели выдающиеся аналогии – стрельба трассирующими, разбитые окна и фантастическое по своей аналогии с управлением вертолетом объяснение необходимости ортогонального подхода, что особенно важно в критической ситуации. Я практически не сомневаюсь, что эта книга станет превосходным источником полезной информации как для начинающих программистов, так и для умудренных опытом мэтров.

Джон Лакос, автор книги Large-Scale C++ Software Design

Когда такие книги появляются на прилавках магазинов, я покупаю по десять экземпляров для раздачи моим клиентам.

Эрик Вот, инженер-программист

Большинство современных книг по разработке программ не в состоянии охватить

основ становления программиста-мастера. Они тратят время на спецификацию синтаксиса или технологии, тогда как на самом деле движущей силой любой команды является наличие талантливых программистов, которые реально владеют своим ремеслом. Отличная книга.

Пит Макбрии, независимый консультант

Прочитав книгу, я реализовал много из тех практических предложений и подсказок, которые дают нам авторы. Честно говоря, они сэкономили моей фирме время и деньги, помогая выполнить работу быстрее! "Программист-прагматик" должен стать настольной книгой для всех, кто зарабатывает на жизнь программированием.

Джаред Ричардсон, старший программист фирмы iRenaissance, Inc.

Я хотел бы, чтобы эта книга попала ко всем новым сотрудникам моей фирмы.

Крис Клилэнд, Старший инженер-программист фирмы Object Computing, Inc.

Предисловие

Книга, которую вы сейчас держите в руках, попала ко мне как рецензенту еще до выхода в свет. Даже в черновом варианте она оказалась превосходной. Дэйву Томасу и Энди Ханту есть что сказать, и они знают, как сказать. Я видел то, над чем они трудились, и уверен, что сделанное ими будет работать. Меня попросили написать это предисловие, в котором я объясняю причины своей уверенности.

В этой книге говорится о способе программирования, которому вы можете последовать. Вполне возможно, что вы даже и не думали, что программирование окажется таким трудным занятием, но дело обстоит именно так. Почему? С одной стороны, не все книги по программированию написаны профессиональными программистами. Многие из них скомпилированы создателями языков программирования или же журналистами, которые сотрудничают с ними в продвижении на рынок этих языков. Книги эти рассказывают вам, как общаться на некоем языке программирования (что, конечно, является немаловажным фактором), но это лишь малая часть того, чем, собственно, занимается программист.

Что же программист делает помимо общения на языке программирования? Эта проблема достаточно глубока. Большинство программистов затруднились бы объяснить, что же они делают. Программирование – это работа, насыщенная подробностями, и для того чтобы уследить за ними, необходимо сосредоточиться. Проходит время, на свет появляется программа. Если всерьез не задумываться над тем, что вы делали, можно придти к выводу, что программирование сводится к набору операторов на специфическом языке. Разумеется, это неправда, но вы ни за что бы так не подумали, осмотревшись по сторонам в секции программирования книжного магазина.

В своей книге "Программист-прагматик" Дэйв и Энди рассказывают нам о способе программирования, которому мы можем последовать. Как же им удалось добиться таких успехов? Не сосредоточились ли они на деталях, уподобившись другим программистам? Нет, они лишь уделили внимание тому, что они делали, во время самой работы, а затем попытались сделать это несколько лучше.

Представьте, что сидите на совещании. Наверное, выдумаете, что совещание длится целую вечность, а вместо него лучше было бы заняться программированием. Дэйв и Энди в такой ситуации думали бы о том, почему происходит это совещание, и задались вопросом, существует ли что-то еще, что они могли бы сделать вместо совещания, и может ли это «что-то» быть автоматизировано таким образом, чтобы это совещание проходило не в настоящем, а в будущем. Затем они бы осуществили задуманное.

Именно таков образ мышления Дэйва и Энди. Это совещание не отвлекало бы их от программирования. Напротив, это и было бы программирование. И этот способ может быть усовершенствован. Я знаю, что они мыслят именно таким образом, поскольку в книге есть подсказка 2: "Думай! О своей работе".

Представьте себе, что авторы мыслят подобным образом на протяжении нескольких лет. У них вскоре должна была бы собраться целая коллекция решений. Теперь представьте, что они используют эти решения в своей работе на протяжении еще нескольких лет и при этом отказываются от слишком трудных решений или тех, что не всегда приводят к желаемому результату. Этот подход и может быть определен как прагматический. Вы, вероятно, подумаете, что подобная информация – настоящая золотая жила. И будете правы.

Авторы рассказывают нам, как они программируют. И рассказывают тем способом, которому мы можем последовать. Но в последнем утверждении есть нечто большее, чем вы

думаете. Позвольте мне объяснить.

Авторы проявили осторожность, избегая выдвижения теории разработки программного обеспечения. Это хорошо, поскольку в противном случае им пришлось бы исказить всю книгу, защищая эту теорию. Подобное искажение является традицией в физике, где теории в конечном счете становятся законами или же преспокойно отвергаются. С другой стороны, программирование подчиняется немногим (если вообще каким-нибудь) законам. Поэтому совет в области программирования, вращающегося вокруг квазизаконов, может прекрасно выглядеть в теории, но на практике не провалиться. Именно это происходит со многим книгами по методологии.

Я изучал эту проблему в течение десяти лет и обнаружил, что самым многообещающим является подход, называемый языком шаблонов. Вкратце шаблон представляет собой некое решение, а язык шаблонов является некой системой решений, подкрепляющих друг друга. Вокруг поиска таких систем сформировалось целое сообщество.

Эта книга – нечто большее, чем просто собрание подсказок. Это и есть язык шаблонов, но в "овечьей шкуре". Я говорю так потому, что каждая подсказка получена из реального опыта, подана как конкретный совет и соотносится с другими, образуя систему. Подсказки представляют собой характеристики, которые позволяют нам изучать язык шаблонов и следовать ему.

Вы можете следовать советам, содержащимся в данной книге, потому что они конкретны. В книге нет расплывчатых абстракций. Дэйв и Энди пишут непосредственно для вас, так, как будто каждая подсказка является жизненно необходимой для пробуждения вашей карьеры в сфере программирования. Они упрощают эту сферу, они рассказывают некую историю, используют легкие намеки, а затем отвечают на вопросы, возникающие, когда вы попробуете сделать что-либо.

Есть и нечто большее. После того как вы прочтете десять или пятнадцать подсказок, вам начнет открываться новое измерение вашей работы. В английском языке это измерение обозначается аббревиатурой QWAN (Quality Without A Name – качество без имени). Книга содержит философию, которая будет внедряться в ваше сознание и смешиваться с вашей собственной. Она не занимается проповедью. Она лишь сообщает, что может работать. Но рассказ способствует проникновению внутрь. В этом состоит красота этой книги: она воплощает философию и делает это непретенциозно.

И вот она перед вами – простая в чтении и применении книга о практике программирования. Я все говорю и говорю о том, почему она действенна. Вам же, вероятно, нужно, чтобы она действовала в принципе. Она действует. Вы это увидите вами.

Уорд Каннингхэм

От авторов

Эта книга поможет вам стать лучшим программистом.

Неважно, кем вы являетесь – разработчиком-одиночкой, членом большой проектной команды или консультантом, одновременно работающим со многими заказчиками. Эта книга поможет вам – отдельно взятой личности – повысить качество работы. Она не посвящена теории, авторы сосредоточились на практических аспектах, на том, как использовать свой опыт для принятия более продуманных решений. Слово «прагматик» происходит от латинского pragmaticus-"сведущий в каком-либо виде деятельности", а оно, в свою очередь, от греческого Trpaxxeiv, означающего "делать что-либо". Таким образом, эта книга посвящена деятельности.

Программирование – это прикладное искусство. Его простейший смысл заключается в следующем: заставить компьютер делать то, что вам нужно (или то, что нужно пользователю, работающему с вашей программой). Программист – он и слушатель, он и советник, он и переводчик и даже диктатор. Вы пытаетесь ухватить суть не совсем ясных требований и найти такой способ их выражения, что только машина сможет оценить его по достоинству. Вы пытаетесь задокументировать работу так, чтобы она была понятна другим, спроектировать ее так, чтобы другие могли на нее положиться. Кроме того, вы пытаетесь сделать все это вопреки безжалостному ходу стрелки часов, отсчитывающих время, отпущенное на проект. Каждый день вы совершаете маленькое чудо.

Это непростая работа.

Многие предлагают вам помощь. Фирмы-поставщики инструментальных средств настойчиво говорят о чудесах, которые творят их программы. Мудрецы от методологии заверяют, что их средства гарантируют результаты. Каждый считает свой язык программирования лучшим из лучших, а операционную систему – панацеей.

Разумеется, эти утверждения неверны. Простых ответов не существует. Нет такого понятия, как наилучшее решение, будь то инструментальное средство, язык или операционная система. Существуют лишь некие системы, которые являются более приемлемыми при конкретном стечении обстоятельств.

И в этот момент на сцену выходит прагматизм. Стоит избегать обряда венчания с конкретной технологией, но при этом необходимо обладать подготовкой и опытом, настолько обширными, что это позволит выбрать верные решения в конкретных ситуациях. Ваша подготовка происходит из понимания основных принципов информатики, а опыт основывается на разнообразных практических проектах. Теория и практика сочетаются, придавая вам силу.

Вы корректируете ваш подход, приспосабливая его к существующим обстоятельствам и окружающей среде. Вы оцениваете относительную важность всех факторов, влияющих на проект, и используете свой опыт в выработке приемлемых решений. И все это делаете непрерывно по ходу работы. Программисты-прагматики делают дело и делают его хорошо.

Кому адресована эта книга?

Эта книга предназначена программистам, желающим повысить эффективность и продуктивность своей работы. Возможно, вы разочарованы тем, что не реализуете до конца свой потенциал. Возможно, смотрите на коллег, которые, как вам кажется, используют инструментальные средства, чтобы опередить вас в продуктивности своего труда. Может быть, в вашей работе используются устаревшие технологии, и вам хотелось бы узнать, как можно приложить новые идеи к тому, над чем вы работаете в данный момент.

Авторы не претендуют на знание ответов на все вопросы (или на большинство из них) и на то, что их идеи применимы к любым ситуациям. Можно лишь сказать, что если следовать подходу авторов, то опыт приобретается быстро, продуктивность увеличивается и целостное понимание процесса разработки программ улучшается. Вы создаете лучший программный продукт.

Как происходит становление программиста-прагматика?

Каждый разработчик уникален, со своими сильными сторонами и слабостями, предпочтениями и неприязнью. С течением времени каждый создает собственную окружающую среду. Эта среда отражает индивидуальность программиста в той же степени, как его (или ее) хобби, одежда или прическа. Однако, если вы принадлежите к программистам-прагматикам, то у вас есть общие черты, характеризующие данный тип:

• **Опережающее восприятие и быстрая адаптация.** У вас есть инстинкт на технологии и методы, и вам нравится проверять их на практике. Вы быстро схватываете новое и объединяете его с уже имеющимися знаниями. Ваша уверенность рождается из опыта.

• **Любознательность.** Вы стремитесь задавать вопросы. "Это здорово – как тебе это удалось?" "У тебя возникали сложности при работе с этой библиотекой?" "Что это за система BeOS, о которой я как-то слышал?" "Как реализуются символические ссылки?" Вы – охотник до мелких фактов, каждый из которых может повлиять на то или иное решение даже годы спустя.

• **Критическое осмысление.** Вы редко принимаете что-то на веру, не ознакомившись предварительно с фактами. Когда коллеги говорят, что "этого не может быть, потому что этого не может быть никогда", или же фирма-поставщик обещает решить абсолютно все ваши проблемы, у вас возникает ощущение близящейся схватки с соперником.

• **Реализм.** Вы пытаетесь нащупать, где же находятся подводные камни в каждой проблеме, с которой приходится сталкиваться. Реализм дает понимание того, насколько трудными могут быть многие предметы и сколько времени займет то или иное действие. Осознание для себя, что процесс должен быть непростым или что для его завершения потребуется время, придаст вам жизненные силы, необходимые для его осуществления.

• **Универсальность.** Вы стараетесь познакомиться с большим числом технологий и операционных систем и работаете, чтобы не отставать от новшеств. Хотя для вашей теперешней работы может потребоваться узкая специализация, вы всегда сможете перейти в новую область, открывая для себя новые горизонты.

Под конец авторы приберегли наиболее общие характеристики. Все программисты-прагматики обладают ими. Они настолько общие, что могут расцениваться как подсказки:

Подсказка 1: Позаботьтесь о вашем ремесле

Нет смысла разрабатывать программы, если вы не заботитесь о качестве работы.

Подсказка 2: Думай! О своей работе

Авторы призывают вас во время работы думать исключительно о работе – только так вы останетесь программистом-прагматиком. Это не случайная оценка существующей практики, а критическая оценка каждого принимаемого вами решения – в течение каждого рабочего дня и по каждому проекту. Никогда не пользуйтесь автопилотом. Думайте постоянно, критикуя свою работу в реальном масштабе времени. Старый девиз фирмы IBM "ДУМАЙ!" является своего рода мантрой для программиста-прагматика.

Если сказанное покажется вам каторжной работой, это значит, что вы обнаруживаете

реалистическое мышление. Это, вероятно, отнимет некоторую часть вашего драгоценного времени – того времени, которое уже спрессовано до крайности. Но наградой станет более активное вовлечение в работу, которую вы любите, чувство властителя над все большим числом предметов и удовольствие, порождаемое чувством постоянного усовершенствования. Вложенное вами время будет приносить доход в течение долгого периода по мере того, как вы и ваша команда будут работать с большей эффективностью, писать программы, которые легче поддерживать, и тратить меньше времени на производственные собрания.

Прагматики-одиночки и большие команды

У некоторых людей возникает чувство, что в больших командах или сложных проектах нет места индивидуальности. "Разработка программ является инженерной дисциплиной, которая нарушается, когда отдельные члены команды начинают решать сами за себя", – говорят они.

Авторы не согласны с этим утверждением.

Разработка программ призвана быть инженерной дисциплиной. Однако это не исключает индивидуального мастерства. Достаточно вспомнить о больших соборах, построенных в Европе в средние века. Для каждого из них потребовались тысячи человеко-лет усилий, прилагаемых на протяжении десятилетий. Приобретенный опыт передавался следующему поколению строителей, которые своими достижениями двигали строительную технику вперед. Но плотники, каменотесы, резчики по дереву и стекольщики оставались мастерами, преобразующими требования для создания единого целого, что выходило за границы чисто механической стороны строительства. Именно вера в их личный вклад не давала замереть этим проектам:

> Отесывая камни, всегда думай о соборах, которые будут строиться из них.
>
> *Кредо средневекового каменотеса*

В общей структуре проекта всегда найдется место индивидуальности и мастерству. Это утверждение особенно верно, если учитывать сегодняшнее состояние программирования. Через сотню лет современные методы программирования могут показаться такими архаичными, какими сегодня кажутся методы строительства средневековых соборов, тогда как наше мастерство по-прежнему будет в почете.

Непрерывность процесса

Во время экскурсии по Итонскому колледжу в Англии турист спросил садовника, как ему удается содержать лужайки в столь идеальном состоянии. "Это несложно, – ответил садовник, – вы просто стряхиваете росу каждое утро, выкашиваете лужайку через день и утрамбовываете раз в неделю".

"И это все?" – спросил турист.

"Абсолютно все, – ответил садовник, – если заниматься этим на протяжении 500 лет, то ваша лужайка будет не хуже".

Великие лужайки, как и великие программисты, нуждаются в ежедневном уходе. В ходе беседы консультанты в области менеджмента не преминут вставить японское слово «кайдзен». "Кайдзен" – японский термин, означающий политику непрерывного внедрения большого количества мелких усовершенствований. Считается, что «кайдзен» стала одной из основных причин резкого роста производительности и качества в японской промышленности, и эту политику стали применять во многих странах. «Кайдзен» применима и к отдельным личностям. Каждый день необходимо работать, оттачивая свои навыки и добавляя в свой репертуар новые произведения. В отличие от итонских газонов, для достижения результата потребуются дни. Годы спустя вы будете поражаться своему преуспеванию и профессиональному росту.

Как составлена эта книга

Книга состоит из кратких разделов, каждый из которых является законченным и посвящен определенной теме. В тексте есть перекрестные ссылки, которые помогают поставить каждую тему в контекст. Разделы можно читать в любом порядке – данная книга не предназначена для чтения от начала до конца.

Периодически вам будут попадаться вставки типа "Подсказка nn" (например, "Подсказка 1: Позаботьтесь о вашем ремесле"). Помимо выделения некоторых важных моментов в тексте, подсказки живут своей собственной жизнью, а авторы живут по ним повседневно.

В приложении A содержится перечень использованных ресурсов: библиографический список, список ссылок на web-ресурсы, а также перечень рекомендованных периодических изданий, книг и профессиональных организаций. В тексте книги есть библиографические ссылки и ссылки на web-ресурсы, такие как [KP99] и [URL 18] соответственно.

Авторы включили также Упражнения и Вопросы для обсуждения. Упражнения (и ответы к ним) как правило, несложные, тогда как Вопросы для обсуждения более замысловаты. Для того чтобы передать свой образ мышления, авторы включили свои собственные ответы к упражнениям в приложение B, но лишь некоторые из упражнений имеют единственное корректное решение. Вопросы для обсуждения могут стать основой для групповых дискуссий или написания эссе на углубленных курсах программирования.

Исходные тексты программ и другие ресурсы

Большинство программ, представленных в этой книге, извлечены из компилируемых исходных файлов, которые можно загрузить с web-сайта www.pragmaticprogrammer.com.

Ваши отклики

Авторам книги интересны ваши отклики. Комментарии, предложения, замеченные в тексте ошибки и проблемы в приведенных примерах всячески приветствуются. Наш электронный адрес:

ppbook@pragmaticptogrammer.com.

Благодарности

Когда мы начали писать эту книгу, у нас и в мыслях не было, сколько коллективных усилий необходимо для ее выпуска в свет.

Издательство Addison-Wesley было как всегда великолепно, пригласив пару начинающих хакеров и показав авторам весь процесс издания книги – от идеи до оригинал-макета. Авторы выражают благодарность Джону Уэйту и Меере Равиндирану за поддержку в начале работы над книгой, Майку Хендриксону, редактору-энтузиасту (и оформителю обложки!), Лоррейн Ферье и Джону Фуллеру за помощь в производстве, а также неутомимой труженице Джулии Дебаггис, связавшей нас воедино.

Затем наступил черед рецензентов. Это Грег Эндресс, Марк Чиэрс, Крис Кли-лэнд, Алистер Кокбэрн, Уорд Каннингхэм, Мартин Фаулер, Тхапг Т. Зиан, Роберт Л.

Гласе, Скотт Хеннингер, Майкл Хантер, Брайан Кирби, Джон Лакос, Пит Макбрин, Кэри П. Моррис, Джаред Ричардсон, Кевин Рулэнд, Эрик Старр, Эрик Ваут, Крис Ван Вик и Дебора Зуковски. Без их заботливых комментариев и ценных советов эта книга читалась бы хуже, была бы менее точной и в два раза длиннее. Благодарим их за уделенное нам время и мудрость.

Второе издание этой книги существенно выиграло за счет пристальных взоров читателей. Благодарим Брайана Блэнка, Пола Боула, Тома Экберга, Брента Фулгэ-ма, Луи Поля Эбера, Хенка-Яна Ульде Лоохюса, Алана Лунда, Гарета Маккофана, Иошики Шибату и Фолькера Вурста за найденные ошибки и деликатность, проявленную при указывании на них авторам.

В течение многих лет мы работали с большим количеством продвинутых клиентов, от них мы набирались опыта, который и описали в этой книге. Недавно мы имели счастье работать с Питером Герке над несколькими проектами. Мы благодарны ему за его поддержку и энтузиазм.

При издании данной книги использовались программные продукты LaTex, pic, Perl, dvips, ghostview, ispell, GNU make, CVS, Emacs, XEmacs, EGCS, GCC, Java, iContract и SmallEiffel, оболочки Bash и zsh в операционной среде Linux. Поражает тот факт, что все эта груда программного обеспечения распространяется абсолютно бесплатно. Авторы говорят «спасибо» тысячам программистов-прагматиков, создавших эти продукты и передавших их нам. Отдельно хотелось бы поблагодарить Рето Крамера за его помощь в работе с iContract.

И последнее, но оттого не менее важное: авторы в огромном долгу перед своими семьями, которые не только смирились с поздним засиживанием за компьютером, огромными счетами за телефонные разговоры и постоянным беспорядком, но и благородно время от времени соглашались прочесть то, что написали авторы. Благодарим их за то, что они не давали нам спускаться с небес на землю.

Энди Хант

Дэйв Томас

Глава 1
Прагматическая философия

Что отличает программистов-прагматиков? Мы полагаем, что это склад ума, стиль, философия подхода к существующим проблемам и их решениям. Прагматики выходят за пределы сиюминутной проблемы, всегда стараются рассмотреть ее в более широком контексте, осознать общую картину происходящего. В конце концов, как можно быть прагматиком вне широкого контекста? Как приходить к интеллектуальным компромиссам и принимать взвешенные решения?

Другим ключом к успеху прагматиков является то, что они берут на себя ответственность за все, что они делают; это обсуждается ниже в разделе "Мой исходный текст съел кот Мурзик". Будучи ответственными, прагматики не сидят, сложа руки, глядя на то, как их проекты погибают из-за небрежного отношения. В разделе "Программная энтропия" говорится о том, как сохранить проекты в первоначальном виде.

Большинство людей с трудом воспринимают изменения: иногда по понятным причинам, иногда в силу старой доброй инерции. В разделе "Суп из камней и сварившиеся лягушки" рассматривается стратегия провоцирования изменений, а также (для равновесия) предлагается поучительная сказка о некоем земноводном, которое не учло опасностей, таящихся в постепенном изменении.

Одним из преимуществ понимания контекста, в котором вы работаете, является более легкое осознание того, насколько хорошими должны быть создаваемые программы. Иногда "почти идеальность" является единственно возможным вариантом, но зачастую приходится идти на компромиссы. Этот аспект исследуется в разделе "Приемлемые программы".

Конечно, необходимо обладать обширной базой знаний и опыта, чтобы все это одолеть. Обучение представляет собой непрерывный и продолжительный процесс. В разделе "Портфель знаний" обсуждаются некоторые стратегии поддержания стремления к обучению.

Разумеется, никто из нас не работает в безвоздушном пространстве. Все мы проводим большое количество времени в общении с другими людьми. В разделе "Общайтесь!" перечислены способы, как сделать это общение более эффективным.

Прагматическое программирование ведет свое начало от философии прагматического мышления. В данной главе приводятся основные положения этой философии.

1
Мой исходный текст съел кот Мурзик

Страх показаться слабым есть величайшая из всех слабостей.

Ж. Б. Боссюэ, Политика и Священное Писание, 1709

Одним из краеугольных камней прагматической философии является идея принятия ответственности за себя и за свои действия с точки зрения карьерного роста, проекта и каждодневной работы. Программист-прагматик принимает на себя ответственность за свою собственную карьеру и не боится признаться в неведении или ошибке. Конечно, это не самый приятный момент программирования, но иногда подобное случается даже с самым лучшим из проектов. Несмотря на тщательное тестирование, хорошее документирование и основательную автоматизацию, все идет не так как надо. Выпуски программ запаздывают. Возникают непредусмотренные технические проблемы.

Подобные вещи случаются, и мы пытаемся справиться с ними настолько профессионально, насколько это возможно. Это означает необходимость быть честным и непосредственным. Мы можем гордиться нашими способностями, но мы должны быть честными, говоря и о наших недостатках – нашем неведении и наших ошибках.

Принятие ответственности

Ответственность – это то, на что активно соглашаются. Вы связываете себя обязательством, чтобы гарантировать, что нечто делается правильно, но ваш непосредственный контроль над каждым аспектом делаемого не является необходимостью. В дополнение к тому, что вы делаете все от вас зависящее, необходимо анализировать ситуацию на предмет наличия неконтролируемых вами рисков. Вы имеет право не принимать на себя ответственность за невозможную ситуацию, или за ситуацию, риски в которой слишком велики. Вам придется сделать самоотвод, основанный на вашей собственной этике и оценках.

Если вы приняли на себя ответственность за результат, то вам придется за него перед кем-то отчитываться. Если вы делаете ошибку (как и все мы), признайте ее честно и попытайтесь предложить варианты исправления.

Не стоит перекладывать вину на кого-либо (или на что-либо) или выдумывать отговорки. Не стоит сваливать все на субподрядчика, язык программирования, менеджмент или коллег по работе. Все они могут сыграть свою роль в неудаче, но вашим делом является решение проблем, а не отговорки.

Если есть вероятность, что субподрядчик не справится со своими обязанностями, то у вас должен быть план на случай возникновения непредвиденных обстоятельств. Если жесткий диск выходит из строя, унося в небытие весь исходный текст, а у вас нет резервной копии, это ваша вина. Фраза "Мой исходный текст съел кот Мурзик", высказываемая вашему шефу, не решит возникшей проблемы.

Подсказка 3: Представьте варианты решения проблемы, а не варианты отговорок

Перед тем как подойти к кому-либо, чтобы высказать, почему что-либо не может быть сделано или уже сломалось, остановитесь и прислушайтесь к себе. Поговорите с резиновым утенком, стоящим на вашем мониторе, или с котом. Как звучит ваша отговорка, разумно или глупо? И как ее воспримет ваш шеф?

Смоделируйте разговор в уме. Что, вероятнее всего, скажет ваш собеседник? Спросит ли он: "А так вы пробовали?" или "А это вы учли?" Как ответить? Перед тем как пойти и сообщить плохие новости, может, попробовать что-то еще? Иногда вы просто знаете, что он собирается сказать, поэтому избавьте его от лишних забот.

Вместо отговорок представьте варианты решения проблемы. Не говорите, что это не может быть сделано, объясните, что может быть сделано для спасения ситуации. Может быть, взять да и выбросить исходный текст? Развивайте эти варианты, используя реорганизацию (см. "Реорганизация"). Стоит ли тратить время на разработку прототипа, чтобы определить лучший способ, который необходимо использовать (см. "Прототипы и памятные записки")? Стоит ли внедрять более совершенные процедуры тестирования (см. "Программа, которую легко тестировать" и "Безжалостное тестирование") или автоматизации (см. "Вездесущая автоматизация"), чтобы предотвратить дальнейшие неудачи? Возможно, вам понадобятся дополнительные ресурсы. Не бойтесь спрашивать или признаться, что нуждаетесь в помощи.

Попытайтесь исключить неубедительные отговорки до того, как их озвучить. Если нужно, выскажите их сначала вашему коту. Ну а если ваш Мурзик возьмет вину на себя…

Другие разделы, относящиеся к данной теме:

- Прототипы и памятные записки
- Реорганизация
- Программа, которую легко тестировать
- Вездесущая автоматизация
- Безжалостное тестирование

Вопросы для обсуждения:

- Как вы отреагируете, когда кто-нибудь – кассир в банке, механик в автосервисе, или клерк придет к вам с подобными отговорками? Что в итоге можно подумать о них лично и об их фирме?

2
Энтропия в программах

Разработка программного обеспечения обладает иммунитетом почти ко всем физическим законам, однако энтропия оказывает на нас сильное влияние. Энтропия – это термин из физики, обозначающий уровень «беспорядка» в системе. К сожалению, законы термодинамики утверждают, что энтропия во вселенной стремится к максимуму. Увеличение степени беспорядка в программах на профессиональном жаргоне называется "порчей программ".

Существует много факторов, вносящих свой вклад в порчу программ. Похоже, что наиболее важным из них является психология или культура в работе над проектом. Даже если команда состоит лишь из одного-единственного сотрудника, психология проекта может быть весьма тонкой материей. При наличии наилучших планов и специалистов экстракласса, проект все же может рухнуть и сгнить в период разработки. Однако существуют и другие проекты, которые, несмотря на огромные трудности и постоянные неудачи, успешно борются с природной тенденцией к беспорядку и заканчиваются хорошо.

В чем же состоит разница?

Некоторые здания, расположенные в старых кварталах города, находятся в хорошем состоянии и чистоте, тогда как другие являют собой жуткие развалины. Почему? Исследователи в области преступности и упадка больших городов открыли удивительный механизм, запускающий процесс быстрого превращения чистенького, нетронутого жилого дома в полуразрушенную и заброшенную трущобу [WK82]

Причина – одно-единственное разбитое окно.

Одно разбитое окно, стекло в котором не меняется в течение длительного времени, развивает в обитателях здания ощущение заброшенности – ощущение, что властям нет дела до того, что происходит со зданием. Затем разбивается другое окно. Люди начинают мусорить. На стенах появляются похабные надписи. Возникают серьезные повреждения строительной конструкции. За относительно короткое время здание портится, несмотря на стремление владельца отремонтировать его, и ощущение заброшенности становится реальностью.

"Теория разбитого окна" дала полицейским участкам в Нью-Йорке и других больших городах стимул: навалиться всем миром на решение малых проблем ради сдерживания больших. И это срабатывает: сосредоточение усилий на первоочередном решении проблем разбитых окон, похабных надписей и других малых правонарушений, привело к сокращению уровня тяжких преступлений.

Подсказка 4: Не живите с разбитыми окнами

Не оставляйте "разбитые окна" (неудачные конструкции, неверные решения или некачественный текст программы) без внимания. Как только вы их обнаружите, чините сразу. Если нет времени на надлежащий ремонт, забейте окно досками. Наверняка вы сможете закомментировать ошибочный фрагмент или вывести на экран сообщение "В стадии разработки", или использовать фиктивные данные. Необходимо предпринять хотя бы малейшее

действие, чтобы предотвратить дальнейшее разрушение, и показать, что вы контролируете ситуацию.

Мы видели, как безошибочные, функциональные системы быстро портились, как только окна начали разбиваться. Существуют и другие факторы, которые вносят свой вклад в порчу программ, и мы коснемся некоторых из них далее, но небрежность ускоряет порчу быстрее, чем любой другой фактор.

Вы можете подумать, что ни у кого не будет времени обойти "разбитые окна" проекта и отремонтировать их. Если вы продолжаете думать подобным образом, тогда вам лучше спланировать приобретение мусорного контейнера или переехать в другой район города. Не давайте энтропии победить себя.

Как погасить пожар

И напротив, существует история абсурдного (до неприличия) опыта одного из авторов книги, Энди Ханта. Его дом был в идеальном порядке, великолепен, наполнен бесценным антиквариатом, произведениями искусства и прочими ценностями. Однажды гобелен, висевший в гостиной слишком близко от камина, загорелся. Пожарные примчались, чтобы спасти положение, а заодно и дом. Но перед тем как втащить в дом свои большие, грязные шланги, они остановились перед полыхающим огнем, чтобы раскатать специальный мат от входной двери до очага пожара. Они боялись испортить ковер.

Конечно, это весьма экстремальный случай, но именно этот способ должен использоваться в случае с программным обеспечением. Одно разбитое окно – неудачно спроектированный фрагмент программы, неверное решение, принятое менеджером (действующее на протяжении всего проекта) – это все, что требуется дня того, чтобы началось отклонение от нормы. Если оказывается, что вы работаете над проектом с несколькими разбитыми окнами, то слишком легко сползти к умонастроению типа "Вся оставшаяся часть программы – это ерунда, я всего лишь следую примеру". Не важно, в каком состоянии находился проект до этого момента. В оригинальном эксперименте, приведшем к возникновению теории разбитых окон, заброшенный автомобиль стоял в течение недели нетронутым. Но как только одно-единственное окно было разбито, автомобиль был «раздет» и перевернут вверх колесами за несколько часов.

К тому же, если вы находитесь в команде и работаете над проектом, тексты программ которого совершенны изначально – корректно написаны, хорошо спроектированы и элегантны – вы, вероятно, предпримете дополнительные усилия к тому, чтобы не испортить их, так как это сделали пожарные с ковром. Даже если речь идет о чрезвычайной ситуации (контрольные сроки, дата выпуска, демонстрационная версия для выставки и т. д.), вы не захотите быть первым среди тех, кто портит проект.

Другие разделы, относящиеся к данной теме:

- Суп из камней и сварившиеся лягушки
- Реорганизация
- Команды прагматиков

Вопросы для обсуждения

• Поспособствуйте укреплению вашей команды, изучив ваших компьютерных «соседей». Выберите одно или два "разбитых окна" и обсудите с вашими коллегами, в чем состоят проблемы и что можно сделать для их решения.

• Можно ли сказать, когда разбивается первое окно? Какова ваша реакция? Если это произошло в результате чьего-либо решения, или по воле руководства, то как вести себя в этом случае?

3
Суп из камней и сварившиеся лягушки

Трое солдат возвращались с войны и проголодались. Когда они увидели впереди деревню, их настроение поднялось – они были уверены, что крестьяне накормят их. Но как только они пришли в деревню, все двери оказались заперты, а окна – закрыты. После долгой войны крестьяне бедствовали и прятали все, что у них есть.

Это не смутило солдат, они вскипятили котел воды и аккуратно положили в него три камня. Удивленные крестьяне вышли посмотреть.

"Это суп из камней", – объяснили солдаты крестьянам. "И это все, что вы в него кладете?" – спросили крестьяне. "Абсолютно все – хотя на вкус он будет намного лучше, если положить в него немного моркови". Один из крестьян убежал и быстро вернулся с корзиной моркови из своего погреба.

Через некоторое время крестьяне вновь спросили: "И это все?"

"Да", – сказали солдаты, "но пара картофелин сделает суп посытнее", И другой крестьянин убежал.

В течение следующего часа солдаты попросили у крестьян другие ингредиенты, которые сделали суп вкуснее: мясо, лук, соль и травы. И каждый раз крестьяне потрошили свои запасы.

Так они сварили большой котел дымящегося супа. Затем солдаты вынули камни и уселись вместе со всей деревней, чтобы поесть досыта – первый раз за многие месяцы.

В истории с супом из камней есть два важных урока. Солдаты обманывали крестьян, используя любопытство последних, чтобы добыть у них пищу. Но что более важно, солдаты явились катализатором, объединяя жителей деревни с тем, чтобы общими усилиями сделать то, что они не смогли сделать сами, – синергетический результат. В конечном итоге выигрывают все.

Иногда в этой жизни вам бы хотелось оказаться на месте солдат.

Вы можете оказаться в ситуации, когда вам точно известно, что нужно сделать и как это сделать. Перед глазами возникает общий план системы, и вы осознаете, что именно так это и должно быть. Но если вы попросите разрешения на проработку аспекта в целом, то столкнетесь с волокитой и пустыми глазами. Люди будут образовывать комиссии, бюджет должен быть одобрен, и все будет усложнено. Каждый будет держаться за свое кресло. Иногда это называется "изначальной усталостью". Пора вытаскивать камни из котла. Выработайте то, о чем вы реально можете попросить. Проработайте детали. Как только вы это сделаете, покажите людям и позвольте им удивиться. Они скажут: "Конечно, было бы лучше, если бы мы добавили". Положим, что это не важно. Расслабьтесь и подождите, пока они не начнут спрашивать вас о добавлении функциональных возможностей, которые вы задумали изначально. Людям легче присоединиться к грядущему успеху. Покажите им свет в конце туннеля, и они сплотятся вокруг вас [1].

Подсказка 5: Будьте катализатором изменений

С другой стороны, история с супом из камней – это история о ненавязчивом и постепенном

обмане. Это история о шорах на глазах. Крестьяне думают о камнях и забывают обо всем остальном в мире. Все мы впадаем в подобное состояние ежедневно. Нечто просто подкрадывается к нам.

Всем нам известны симптомы. Проекты медленно и неизбежно полностью выходят из-под контроля. Большинство программных катастроф начинаются с малозаметных вещей, и большинство проектов в один прекрасный день идут вразнос. Шаг за шагом система отклоняются от требований, при этом фрагмент текста программы обрастает «заплатами», пока от оригинала не остается ничего. Зачастую именно скопившиеся мелочи приводят к разрушению морали и команд.

Подсказка 6: Следите за изменениями

Сами мы, по чести, никогда этого не делали. Но говорят, что если взять лягушку и бросить ее в кипящую воду, то она сразу выпрыгнет наружу. Однако если бросить лягушку в кастрюлю с холодной водой, а затем медленно нагревать ее, то лягушка не заметит медленного увеличения температуры и останется сидеть в кастрюле, пока не сварится.

Заметим, что проблема лягушки отличается от проблемы разбитых окон, обсуждаемой в разделе 2. В "теории разбитых окон" люди теряют волю к борьбе с энтропией, поскольку она никого не волнует. Лягушка же просто не замечает изменений.

Не будьте лягушкой. Не сводите глаз с общей картины происходящего. Постоянно наблюдайте за тем, что происходит вокруг вас, а не только за тем, что делаете вы лично.

Другие разделы, относящиеся к данной теме:

- Энтропия в программах
- Программирование в расчете на совпадение
- Реорганизация
- Западня требований
- Команды прагматиков

Вопросы для обсуждения

• Просматривая черновик данной книги, Джон Лакос поднял следующий вопрос: солдаты постоянно обманывали крестьян, но в результате изменений, катализатором которых они стали, лучше стало всем. Однако, постоянно обманывая лягушку, вы наносите ей вред. Когда вы пытаетесь ускорить изменения, то можете ли определить, варите вы суп из камней или же лягушку? Является ли это решение субъективным или объективным?

4
Приемлемые программы

Для лучшего добро сгубить легко.

У. Шекспир, Король Лир, действие 1, сцена 4

Существует старый анекдот об американской фирме, которая заказала 100000 интегральных схем на предприятии в Японии. В условиях контракта указывался уровень дефектности: один чип на 10000. Несколько недель спустя заказ прибыл в Америку: одна большая коробка, содержащая тысячи интегральных схем, и одна маленькая, в которой находилось десять схем. К маленькой коробке был приклеен ярлычок с надписью "Дефектные схемы".

Если бы у нас был такой контроль качества! Но реальный мир не позволяет производить многое из того, что является действительно совершенным – особенно программы без ошибок. Время, технология и темперамент – все находится в сговоре против нас.

Однако это не должно вас обескураживать. По словам Эда Иордона, автора статьи в журнале IEEE Software [You95], вы можете обучиться созданию приемлемых программ – приемлемых для ваших пользователей, служб сопровождения и с точки зрения вашего же собственного спокойствия. Вы обнаружите, что производительность вашего труда повысилась, а ваши пользователи стали чуть-чуть счастливее. Кроме того, ваши программы станут лучше за счет сокращения инкубационного периода.

Перед тем как пойти дальше, мы должны определиться в том, что собираемся сказать. Фраза «приемлемый» не подразумевает неаккуратную или неудачно написанную программу. Все удачные системы должны отвечать требованиям их пользователей. Мы просто призываем к тому, чтобы пользователям была дана возможность участвовать в процессе принятия решения, если созданное вами действительно приемлемо.

Находите компромисс с пользователями

Обычно вы пишете программы для других людей. Часто вы вспоминаете о том, что хорошо бы получить от них требования [2]. Но как часто вы спрашиваете их, а насколько хорошими они хотят видеть эти программы? Иногда выбирать не из чего. Если вы работаете над передовыми технологиями, космическим челноком, или низкоуровневой библиотекой, которая будет широко распространяться, то требования будут более строгими, а варианты – ограниченными. Но если вы работаете над новым продуктом, то у вас будут ограничения другого рода. Маркетологам придется сдерживать обещания, вероятные конечные пользователи могут строить планы, основанные на дате поставки программы, а ваша фирма, конечно, будет ограничена в денежных средствах. Профессионалы не могут игнорировать требования пользователей – просто добавить к программе новые средства или «отшлифовать» еще раз тексты программ. Мы не призываем к паническим настроениям: одинаково непрофессионально обещать невероятные сроки и срезать основные "технические углы" чтобы уложиться вовремя.

Сфера действия и качество создаваемой вами системы должны указываться в части системных требований.

Подсказка 7: Сделайте качество одним из пунктов требований

Часто вы будете оказываться в ситуациях, когда необходимо идти на компромисс. Удивительно, но многие пользователи предпочтут использовать программы с некоторыми недоработками, но сегодня, чем год ожидать выпуска мультимедийной версии. Многие IT-департаменты, имеющие ограничения по бюджету, могли бы согласиться с этим утверждением. Хорошие программы (но сегодня) зачастую являются более предпочтительными по сравнению с отличными программами (но завтра). Если вы заранее дадите другим пользователям поиграться с вашей программой, то часто их отзывы будут способствовать выработке лучшего конечного решения (см. "Стрельба трассирующими").

Знайте меру

В ряде случаев программирование подобно живописи. Вы начинаете с чистого холста и определенных базовых исходных материалов. Используете сочетание науки, искусства и ремесла, чтобы определить, что же делать с ними. Набрасываете общую форму, выписываете основу, затем работаете над деталями. Постоянно отступаете назад, чтобы критически взглянуть на то, что же вы сделали. Иногда отбрасываете холст и начинаете снова.

Но художники скажут вам, что вся тяжелая работа идет насмарку, если вы не знаете, в какой момент нужно остановиться. Если вы добавляете слой за слоем, деталь за деталью, живопись может потеряться в краске.

Не стоит портить очень хорошую программу путем приукрашивания и излишней шлифовки. Двигайтесь вперед и дайте вашей программе отстаивать свои права в течение какого-то времени. Она может быть несовершенной. Не беспокойтесь, возможно, она никогда не станет совершенной. (В главе 6 мы обсудим философию разработки программ в несовершенном мире.)

Другие разделы, относящиеся к данной теме:

- Стрельба трассирующими
- Западня требований
- Команды прагматиков
- Большие надежды

Вопросы для обсуждения

• Обратите внимание на производителей инструментальных программных средств и операционных систем, которыми вы пользуетесь. Можете ли вы найти свидетельство тому, что эти компании не испытывают неудобства, поставляя программное обеспечение, хотя им известно, что оно несовершенно? Как пользователь, вы скорее: (1) подождете, пока они устранят все ошибки, (2) выберете усложненную версию программы и примете отдельные ошибки или (3) выберете упрощенную версию программы, но с меньшим числом дефектов?

• Рассмотрите эффект разбиения на модули при поставке программного обеспечения. Больше или меньше времени потребуется для доведения монолитного программного блока до требуемого уровня качества по сравнению системой, спроектированной по модульному принципу? Можете ли вы привести коммерческие примеры?

5
Портфель знаний

Инвестиции в знания окупаются лучше всего.

Бенджамин Франклин

Ах, старина Франклин! Никогда не лез в карман за многозначительным наставлением. Если бы мы рано ложились и рано вставали, мы стали бы великими программистами, не так ли? Ранняя птичка никогда не остается без червячка, но что при этом происходит с червячком?

Хотя в данном случае Бенджамин действительно попал в точку. Знание и опыт являются самыми важными профессиональными активами.

К сожалению, знания и опыт представляют собой истекающие активы [3]. Ваше знание устаревает по мере того, как разрабатываются новые методики, языки, технологии и операционные среды. Изменение расстановки сил на рынке может сделать ваш опыт устаревшим или полностью неприменимым. Принимая во внимание скорость, с которой промчались годы Интернета, это может произойти довольно быстро.

По мере того как величина ваших знаний уменьшается, то же самое происходит с ценностью вас для фирмы-работодателя или заказчика. Мы хотели бы предотвратить возникновение подобной ситуации.

Ваш портфель знаний

Портфелями знаний мы предпочитаем называть все факты, известные программистам об информатике, области приложений, в которых они работают, и накопленный ими опыт. Управление портфелем знаний очень похоже на управление финансовым портфелем:

1. Серьезные инвесторы инвестируют регулярно – это как привычка.
2. Диверсификация – это залог успеха в течение длительного времени.
3. У проворных инвесторов портфель всегда сбалансирован – в нем имеются и консервативные, и высокорисковые, высокодоходные инвестиции.
4. Инвесторы стараются покупать ценные бумаги подешевле и продавать их подороже, обеспечивая тем самым максимальный возврат.
5. Портфели нуждаются в периодическом пересмотре и повторной балансировке.

Управляйте вашим портфелем знаний, используя те же самые принципы, и ваша карьера будет успешной.

Построение вашего портфеля

• **Инвестируйте на регулярной основе.** Как и в случае финансов, необходимо регулярно инвестировать в ваш портфель знаний. Даже если объем инвестиций невелик, сама по себе привычка важна, как, впрочем, и объемы. Несколько примеров на эту тему приводятся в следующем разделе.

• **Инвестируйте в различные сферы.** Чем больше вы знаете о различных вещах, тем большую ценность вы представляете. Как минимум вы обязаны знать плюсы и минусы конкретной технологии, с которой вы работаете в данный момент. Но не останавливайтесь на этом. Лицо информатики меняется быстро – новейшая технология сегодняшнего дня может

оказаться почти бесполезной (или, по меньшей мере, не найти спроса) завтра. Чем больше технологий вы освоите, тем легче вам будет приспособиться к изменениям.

• **Управляйте риском.** Технология находится в некоем диапазоне – от рисковых и потенциально высокодоходных до низкорисковых и низкодоходных стандартов. Вложение всех ваши денег в высокорисковые акции, курс которых может внезапно обвалиться, и другая крайность – консервативное вложение и упущение возможностей – не самые лучшие идеи. Не кладите все "технические яйца" в одну корзину.

• **Покупайте подешевле, продавайте подороже.** Обучение передовой технологии до того, как она станет популярной, может быть столь же сложной задачей, как найти обесцененные акции, но отдача может стать наградой. Изучение языка Java, когда он только что появился, могло показаться рискованным, но оно щедро вознаградило тех, кто принял это раньше всех, и сегодня они занимают лидирующие позиции в данной области.

• **Пересмотр и повторная балансировка.** Информатика – очень динамичная отрасль. Новейшая технология, которую вы начали изучать в прошлом месяце, сегодня может устареть. Возможно, вам понадобится восстановление навыков по технологии баз данных, которой вы не пользовались какое-то время. А может быть, вы смогли бы стать лучшей кандидатурой на открывшуюся вакансию, если бы попробовали изучить другой язык…

Из всех этих директив, самой важной и самой простой в исполнении является

Подсказка 8: Инвестируйте регулярно в ваш портфель знаний

Цели

Теперь у вас есть некоторые директивы, что и когда добавлять к вашему портфелю знаний, как лучше приобрести интеллектуальный капитал, который будет вложен в ваш портфель? Вот несколько предложений.

• **Учите (как минимум) по одному языку программирования каждый год.** Различные языки решают различные проблемы по-разному. Выучив несколько различных подходов, вы можете расширить мышление и избежать закоснелости. Вдобавок, изучать многие языки сейчас намного легче, благодаря богатому выбору бесплатно распространяющегося программного обеспечения в сети Интернет (см. Приложение А).

• **Читайте по одной технической книге ежеквартально.** В книжных магазинах полным-полно технической литературы по темам, интересующим вас или связанным с проектом, над которым вы работаете в настоящее время. Как только это войдет у вас в привычку, читайте по одной книге в месяц. После того как вы овладеете технологиями, которыми вы пользуетесь на данный момент, расширяйте круг своих интересов и изучайте другие технологии.

• **Читайте книги, не относящиеся к технической литературе.** Важно помнить, что пользователями компьютеров являются люди – люди, чьи потребности вы пытаетесь удовлетворить. Не забывайте о человеческом факторе.

• **Повышайте квалификацию на курсах.** Ищите интересные курсы в вашем районе, школе или университете, а может быть, и во время грядущей технической выставки, которая проводится в вашем городе.

• **Участвуйте в собраниях локальных групп пользователей.** Но не просто приходите и слушайте, а принимайте активное участие. Изоляция может оказаться смертельной для вашей

карьеры; разузнайте, над чем работают люди за пределами вашей компании.

• **Экспериментируйте с различными операционными средами.** Если вы работали только в среде Windows, поиграйте со средой Unix дома (для этой цели прекрасно подходит бесплатно распространяемая версия Unix). Если вы использовали только сборочные файлы и редактор, попробуйте интегрированную среду разработчика и наоборот.

• **Оставайтесь в курсе событий.** Подпишитесь на профессиональные журналы и другие периодические издания (рекомендации приведены в Приложении А). Выберите из них те, которые покрывают технологии, отличные от вашего текущего проекта.

• **Подключайтесь к информационным сетям.** Хотите знать плюсы и минусы нового языка или технологии? Группы новостей отлично подходят для поиска практических результатов работы с ними других людей, используемого ими жаргона и т. д. Походите по Интернету в поисках статей, платных сайтов, и любых других доступных источников информации.

Важно продолжать инвестирование. Как только вы почувствуете, что освоили новый язык или фрагмент технологии, двигайтесь дальше. Изучайте другой.

Неважно, будете ли вы когда-либо использовать одну из этих технологий в проекте, или даже не упомянете о них в своем резюме. Процесс обучения расширит ваше мышление, открывая для вас новые возможности и новые пути в творчестве. "Перекрестное опыление" идей важно; попытайтесь применить выученные уроки к проекту, над которым вы работаете в настоящее время. Даже если в вашем проекте не используется некая технология, вы наверняка сможете позаимствовать некоторые идеи. К примеру, ознакомьтесь с объектно-ориентированным подходом, и вы напишете простые программы на языке C различными способами.

Возможности обучения

Итак, вы жадно и много читаете, находитесь в курсе всех новейших разработок в вашей сфере (это не так-то легко) и кто-то задает вам вопрос. У вас нет даже намека на идею, каким должен быть ответ, но вы не признаете это открыто, как и многие.

В этот момент не останавливайтесь. Примите это как брошенный вам вызов для поиска ответа. Спросите гуру (если в вашем офисе нет гуру, вы должны найти его в Интернете: см. следующую врезку.) Поищите в Интернете. Сходите в библиотеку [4].

Если вы не можете найти ответ самостоятельно, найдите того, кто это может. Не бросайте поиски. Разговор с другими людьми поможет в построении вашей собственной сети, и вы можете удивиться, находя по пути ответы на другие, не относящиеся к делу проблемы. И этот старый портфель все утолщается и утолщается…

Все это чтение и исследование требует времени, а времени уже не хватает. Так что вам придется планировать наперед. Запаситесь литературой на то время, которое может бездарно пропасть. Время, которое проходит в очередях на прием к врачам, можно с пользой потратить на чтение литературы – но убедитесь, что вы принесли с собой ваш журнал, а не замусоленную страницу из газеты 1973 года о положении в Папуа Новой Гвинее.

Критическое осмысление

Последним важным пунктом является критическое осмысление того, что вы прочли или услышали. Необходимо убедиться, что знание в вашем портфеле является точным и не поддается влиянию субподрядчика или типа носителя информации. Опасайтесь фанатиков, настаивающих на том, что их догма обеспечивает единственно правильный ответ, – последний может быть

применим или неприменим к вам и вашему проекту.

Всегда имейте в виду силу меркантильности. Первое попадание, выданное поисковой системой, не обязательно оказывается наилучшим; владелец содержимого может просто заплатить, чтобы оказаться в начале списка. Если книжный магазин «раскручивает» книгу, это вовсе не означает, что она хороша, или даже популярна; за это просто могли заплатить.

Подсказка 9: Критически анализируйте прочитанное и услышанное

К сожалению, простых ответов немного. Но, обладая растущим портфелем и применив некоторый критический анализ к потоку изучаемой вами технической литературы, вы сможете понять и сложные ответы.

Уход за гуру и их разведение

С глобальным принятием сети Интернет, гуру внезапно стали ближе – на расстоянии нажатия клавиши Enter. Итак, как найти гуру и вызвать его на разговор?

Здесь есть несколько простых уловок.

• Знайте точно, что вы хотите спросить, и будьте конкретным, насколько это возможно.

• Формулируйте ваш вопрос внимательно и вежливо. Помните, что вы просите одолжения; в противном случае может показаться, что вы требуете ответа.

• Как только вы сформулировали вопрос, остановитесь и вновь поищите ответ. Выхватите несколько ключевых слов и поищите их в Интернете. Поищите подходящие списки часто задаваемых вопросов и ответов на них.

• Решите, каким образом вы зададите вопрос: в открытой форме или же частным образом. Группы новостей Usenet – прекрасное место встреч для экспертов практически по любой теме, но некоторые опасаются открытого характера этих групп. Кроме того, вы всегда можете отправить сообщение непосредственно вашему гуру по электронной почте. В любом случае используйте строку темы сообщения со смыслом. (Сообщение "Нужна помощь!" не останется незамеченным.)

• Расслабьтесь и наберитесь терпения. Люди заняты, и, возможно, потребуется несколько дней, чтобы получить конкретный ответ.

И наконец, обязательно поблагодарите всех, кто ответил вам. И если вы видите людей, задающих вопросы, на которые вы можете ответить, ответьте взаимностью и примите участие.

Вопросы для обсуждения

• На этой неделе начните учить новый язык программирования. Всегда программировали на C++? Попытайтесь выучить язык Smalltalk [URL 13] или Squeak [URL 14]. Работаете с Java? Попробуйте поработать с языком Eiffel [URL 10] или TOM [URL 15]. Информация о других бесплатных компиляторах и средах разработчиков содержится в Приложении A.

• Начните читать новую книгу (но сначала прочтите эту книгу до конца!) Если вы занимаетесь детальной реализацией и программированием, прочтите книгу по проектированию и архитектуре. Если вы занимаетесь высокоуровневым проектированием, прочтите книгу о

методиках программирования.

• Найдите время для разговора о технологии с людьми, которые не участвуют в проекте, над которым вы работаете в настоящее время, или с теми, кто не работает в вашей фирме. Общение может проходить в кафетерии вашей фирмы, а может быть, стоит поискать коллег-энтузиастов на собрании локальной группы пользователей.

6
Общайтесь!

Лучше быть проигнорированным вовсе, чем недооцененным.

Мэй Уэст, Красавица 90-х, 1934.

Может быть, мы способны выучить урок, преподанный мисс Уэст. Важно не только то, что у вас есть, но и как оно упаковано. Лучшие идеи, лучшие программы или самое прагматичное мышление практически не приносят результата, если вы не можете общаться с другими людьми. Без эффективного общения удачная идея может осиротеть.

Нам, разработчикам, приходится общаться на многих уровнях. Мы проводим время на собраниях, слушаниях и переговорах. Мы работаем с конечными пользователями, пытаясь понять их нужды. Мы пишем программы, которые передают наши намерения машине и документируют наши размышления для будущих поколений разработчиков. Мы пишем предложения и служебные записки, в которых требуем и обосновываем предоставляемые нам ресурсы, сообщая наш статус и предлагая новые подходы. Мы работаем ежедневно в своих командах, отстаивая наши идеи, изменяя существующую практику и вводя новую. Большая часть дня проходит в общении, поэтому нам необходимо овладеть его искусством.

Мы обобщили ряд идей, которые находим полезными.

Знайте то, что вы хотите сказать

Возможно, самой трудной частью более формальных стилей общения, используемых в бизнесе, является выработка именно того, что вы хотите сказать. Беллетристы в деталях намечают сюжет книги перед тем, как начать свой труд, но люди, составляющие технические документы, часто рады сесть за клавиатуру, напечатать "1. Введение" и затем набирать все, что прилет им в голову.

Планируйте то, что вы хотите сказать. Напишите «рыбу». Затем спросите себя: "Не противоречит ли это тому, что я пытаюсь высказать?" Совершенствуйте содержание, пока не наступит момент выступления.

Этот подход применим не только к написанию документов. Когда вы готовитесь к важной встрече или телефонному разговору с важным заказчиком, законспектируйте идеи, которыми собираетесь обменяться, и разработайте несколько стратегий для четкого их изложения.

Знайте вашу аудиторию

Вы общаетесь только в том случае, если передаете информацию. Для этого вам необходимо осознавать потребности, интересы и способности вашей аудитории. Всем нам приходилось присутствовать на собраниях, где нахал-разработчик затуманивает глаза вице-президенту по маркетингу долгим монологом о заслугах некоторой скрытой технологии. Это не общение: это просто разговоры и это утомляет [5].

Составьте устойчивый психологический образ вашей аудитории. Акростих WISDOM, показанный на рисунке 1.1, может помочь в этом.

Например, вы собираетесь предложить интернет-систему, позволяющую конечным пользователям представлять отчеты об ошибках. Об этой системе можно рассказать по-разному, в зависимости от аудитории. Конечные пользователи обрадуются тому, что смогут представлять отчеты об ошибках 24 часа в сутки, не занимая телефона. Отдел маркетинга сможет использовать этот факт в целях увеличения объема продаж. Менеджеры в отделе поддержки будут счастливы по двум причинам: можно будет обойтись меньшим числом сотрудников и генерация отчетов о возникающих проблемах будут автоматизирована. И, наконец, разработчики смогут приобрести опыт в работе с клиент-серверными интернет-технологиями и новым ядром баз данных. Выступая перед каждой из этих групп с отдельной трибуны, вы добьетесь того, что они станут неравнодушными к вашему проекту.

Выбирайте подходящий момент

Итак, наступила пятница, конец рабочего дня, неделю назад на фирме прошла аудиторская проверка. Младший ребенок вашей начальницы попал в больницу, на улице идет проливной дождь, и дорога домой представляется сущим кошмаром. Не самое лучшее время для просьб о наращивании памяти на вашем компьютере.

Необходимо уяснить для себя приоритеты аудитории, чтобы лучше понять то, что она хочет услышать от вас. Поймайте менеджера, получившего выговор от шефа, потому что потерялась часть исходного текста программы, и вы найдете в его лице слушателя, который лучше воспринимает ваши идеи о централизованных БД данных исходных текстов. То, что вы говорите, должно быть уместным по времени и содержанию. Иногда для этого достаточно лишь задать вопрос типа: "Удобно ли сейчас поговорить о…?"

What do you want them to learn? (Чему вы хотите их научить)
What is their interest in what you have got to say? (Какова их заинтересованность в вашей речи?)
How sophisticated are they? (Насколько искушена ваша аудитория?)
How much detail do they want? (Насколько детальным должно быть выступление?)
Whom do you want to own the information? (Кто должен обладать информацией?)
How can you motivate them to listen to you? (Как мотивировать слушателей?)
Буквы оригинала складываются в слово «Wisdom» – мудрость (англ.)

Рис. 1.1. Акростих WISDOM – о понимании аудитории

Выбирайте стиль

Определите стиль подачи материала в соответствии с требованиями аудитории. Одним хочется формального брифинга в стиле "только факты". Другим нравятся долгие, обширные беседы, перед тем как перейти к делу. Что касается печатных материалов, то одни любят получать большие переплетенные отчеты, тогда как другие ожидают простой записки или сообщения по электронной почте. Если сомневаетесь, спрашивайте.

Однако следует помнить, что вы – лишь одна половина коммуникационной транзакции. Если кто-нибудь говорит, что ему необходимо описать что-либо в одном абзаце, а вы видите, что это можно сделать лишь в объеме нескольких страниц, скажите ему об этом. Помните, этот вид

отклика также является формой общения.

Встречают по одежке

Ваши идеи важны. Они заслуживают хорошего способа их подачи вашей аудитории.

Многие разработчики (и их менеджеры) при подготовке письменных документов сосредоточены исключительно на содержании. Мы думаем, что это ошибочно. Любой шеф-повар скажет вам, что вы можете корпеть на кухне часами и затем обратить в прах все ваши усилия неправильной сервировкой стола.

Сегодня нет оправдания подготовке небрежных печатных материалов. Современные текстовые процессоры (наряду с настольными издательскими системами, такими как LaTeX и troff) могут печатать великолепные выходные документы. Вам необходимо выучить лишь несколько основных команд. Если ваш текстовый процессор поддерживает стили, используйте их. (Ваша фирма могла уже определить стили, которыми вы можете пользоваться). Выучите, как задаются верхние и нижние колонтитулы. В поисках идей стиля и макета, взгляните на образцы документов, включенные в текстовый процессор. Проверьте правописание, вначале автоматически и затем вручную. *Ведь в право писании мокнут встретить си и такие ушиб кий, кто торты программа не смолит у ловить.*

Привлекайте свою аудиторию

Мы часто обнаруживаем, что документы, которые мы составляем, менее важны, чем процесс их составления. Если возможно, привлекайте ваших читателей с момента появления черновиков документов. Получите их отклики и используйте их идеи. Вы построите хорошие рабочие взаимоотношения и наверняка улучшите составленный документ.

Умейте слушать

Существует одна методика, которую вы должны использовать, если хотите, чтобы люди слушали вас: прислушивайтесь к ним. И в ситуации, когда вы располагаете всей информацией, и во время формального собрания, на котором выдержите речь перед двадцатью руководителями – если вы не будете слушать их, они не будут слушать вас.

Вдохновляйте людей завязать беседу, задавая вопросы или заставляя их подытожить сказанное вами. Превратите собрание в диалог, и вы сможете выделить что-либо более эффективно. Может быть, вы почерпнете что-то и для себя.

Обращайтесь к людям

Если вы задаете кому-нибудь вопрос, а вам не отвечают, то вы полагаете, что данный человек невежлив. Но как часто вы не можете ответить людям, когда они посылают вам сообщение по электронной почте или служебную записку, пытаясь получить информацию, или требуют какого-либо действия? Всегда отвечайте на сообщения электронной почты и голосовые сообщения, даже если ваш ответ звучит просто: "Я вернусь к вам с этим позже". Если вы

держите людей в курсе, они намного легче прощают случайные промахи, и чувствуют, что о них не забыли.

Подсказка 10: Важно, что говорить и как говорить

Поскольку вы не работаете в безвоздушном пространстве, вам необходимо уметь общаться. Чем эффективнее это общение, тем более влиятельным вы становитесь.

Связь по электронной почте

Все, что сказано о коммуникации в письменном виде, одинаково применимо и к электронной почте. Электронная почта стала основой внутрикорпоративных и межкорпоративных коммуникаций. Электронная почта используется при обсуждении контрактов, решении споров и в качестве свидетельства в суде. Но, в силу некоторых причин, люди, которые никогда бы не выслали убогий бумажный документ, позволяют себе распространять отвратительного вида сообщение по всему миру.

Наши подсказки относительно электронной почты довольно просты:

• Перед тем как щелкнуть мышкой на кнопке SEND (Отправить), тщательно проверьте текст.

• Проверьте правописание.

• Используйте простой формат. Некоторые люди читают сообщения электронной почты, используя пропорциональные шрифты, так что картинки, которые вы старательно создавали при помощи символов ASCII, для них будут выглядеть так, как будто это писала "курица лапой".

• Используйте форматы RTF или HTML, если вы точно знаете, что все получатели смогут прочесть послание. Простой текст универсален.

• Старайтесь сводить цитирование к минимуму. Никто не любит получать назад свое собственное сообщение в 100 строк, снабженное пометкой "согласен".

• Если вы цитируете сообщения других людей, убедитесь, что они атрибутированы, и цитируйте их в тексте (это лучше, чем вложение).

• Не используйте обидных сообщений, если не хотите, чтобы позже они вернулись, лишив вас покоя.

• Перед отправкой необходимо проверить список адресатов. Статья в недавнем номере "Уолл-стрит джорнэл" рассказывает о служащем, который взялся распространять критические высказывания о своем шефе по отделу, не подумав о том, что адрес шефа был включен в список рассылки.

• Архивируйте и организуйте вашу электронную почту – как получаемые, так и отсылаемые материалы.

Как обнаружили служащие фирм Microsoft и Netscape во время расследования Министерства юстиции США (1999 г.), электронная почта – это бессмертно. Постарайтесь заботиться об электронной почте так, как вы заботитесь о любой написанной записке или отчете.

Другие разделы, относящиеся к данной теме:

• Прототипы и памятные записки
• Команды прагматиков

Вопросы для обсуждения

• Есть несколько хороших книг, описывающих взаимодействие внутри команд разработчиков [Bro95, McC95, DL99]. Следует обратить на это особое внимание и попытаться прочесть все три книги течение следующих 18 месяцев. В дополнение к этому, книга "Dinosaur Brains" [Вег96] посвящена эмоциональному багажу, который мы вносим в рабочую среду.

• В следующий раз, когда вам придется проводить презентацию или писать служебную записку, отстаивающую некую позицию, до начала работы воспользуйтесь акростихом WISDOM. Посмотрите, поможет ли это вам в представлении того, с чем вы выступаете. Если это возможно, поговорите со своей аудиторией после выступления, и посмотрите, насколько точной оказалась ваша оценка их потребностей.

Глава 2
Прагматический подход

Существует ряд подсказок и уловок, применимых ко всем уровням разработки программ: идеи, которые почти аксиоматичны, и процессы, которые практически универсальны. Однако эти подходы редко документируются как таковые; в основном они фиксируются как случайные высказывания в дискуссиях по проектированию, руководству проектами или программированию.

В этой главе эти идеи и процессы сводятся воедино. Первые два раздела, "Пороки дублирования" и «Ортогональность», тесно связаны между собой. Первый предостерегает от дублирования знания в ваших системах, второй – от растаскивания единого фрагмента знания по многим компонентам системы.

Все вокруг меняется очень быстро, и становится все труднее и труднее поддерживать приложения на должном уровне. В разделе «Обратимость» рассматриваются некоторые методики, позволяющие изолировать проекты от изменяющейся окружающей среды.

Следующие два раздела также связаны между собой. В разделе "Стрельба трассирующими" говорится о стиле разработки программ, позволяющем одновременно осуществлять сбор требований, тестировать проектные решения и реализовывать текст программы. Если для вас это звучит слишком хорошо, чтобы быть правдой, то так оно и есть: разработки в стиле "Стрельба трассирующими" применимы не всегда. Для последнего случая в разделе "Прототипы и памятные записки" показано, как при тестировании архитектур, алгоритмов, интерфейсов и идей используются прототипы.

По мере того как информатика становится зрелой наукой, разработчики изобретают языки программирования все более высокого уровня. Поскольку компилятор, работающий по принципу "сделай так, как приказано" еще не изобретен, в разделе "Языки, отражающие специфику предметной области" представлен ряд более скромных предложений, которые можно реализовать для себя.

Ну и наконец, все мы работаем в мире с ограниченным временем и ресурсами. Оба этих недостатка переживаются легче (радуя ваше начальство), если поднатореть в оценке продолжительности какого-либо дела, о чем и говорится в разделе "Оценка".

Держа в голове эти фундаментальные принципы, можно создать программу, которая будет лучше, быстрее и устойчивее. Ее можно даже сделать на вид более простой.

7
Пороки дублирования

Капитан Джеймс Т. Кирк больше всего любил отключать хищный искусственный интеллект, вводя в компьютер два противоречащих друг другу фрагмента знания. К несчастью, этот принцип оказывается столь же эффективным при доведении вашей программы до обморочного состояния.

Программисты собирают, организуют, сопровождают и связывают воедино знание. Знание документируется в требованиях, воплощается в запускаемых программах и используется для контроля в ходе тестирования.

К сожалению, знание нестабильно. Оно изменяется – часто очень быстро. Понимание некоего требования может измениться после встречи с заказчиком. Правительство изменяет административные положения, и некая бизнес-логика устаревает. Тесты могут показать, что выбранный алгоритм не будет работать. Вся эта нестабильность означает, что мы проводим большую часть времени в режиме сопровождения, осуществляя реорганизацию знания и выражая его по-новому в опекаемых нами системах.

Большинство людей полагает, что сопровождение начинается в момент выпуска приложения в свет и означает устранение ошибок и улучшение характеристик. Мы думаем, что эти люди ошибаются. Программисты постоянно находятся в режиме сопровождения. Наше понимание изменяется день ото дня. Новые требования возникают по мере того, как мы проектируем или создаем текст программы. Возможно, изменяется операционная система. Какой бы ни была причина, сопровождение является не дискретным видом деятельности, а рутинной частью процесса разработки в целом.

Когда мы осуществляем сопровождение, нам приходится отыскивать и изменять представления о предметах – капсулах знания, заложенных в приложение. Проблема состоит в том, что тиражировать знание в требованиях, процессах и программах, которые мы разрабатываем, легко, и когда мы поступаем подобным образом, возникает призрак сопровождения – тот самый, который начинает делать свое черное дело задолго до отправки готового приложения заказчику.

Мы полагаем, что единственно надежным способом разработки программ и облегчения их понимания и сопровождения является следование принципу "Не повторяй самого себя" (Далее DRY = Don't Repeat Yourself. – Прим. пер.):

КАЖДЫЙ ФРАГМЕНТ ЗНАНИЯ ДОЛЖЕН ИМЕТЬ ЕДИНСТВЕННОЕ, ОДНОЗНАЧНОЕ, НАДЕЖНОЕ ПРЕДСТАВЛЕНИЕ В СИСТЕМЕ.

Подсказка 11: Не повторяй самого себя

Альтернативой является представление одного и того же предмета в двух или более местах. Если меняется одно, придется вспоминать и об изменении других, или же ваша программа (подобно компьютерам пришельцев) будет поставлена на колени в виду противоречий. Вопрос не в том, вспомните ли вы о необходимом изменении или нет; вопрос в том, когда вы об этом забудете.

Вы обнаружите, что принцип DRY будет время от времени появляться на протяжении всей книги, часто в контексте, который не имеет ничего общего с программированием. Мы полагаем,

что этот принцип является одним из наиболее важных инструментов в арсенале программиста-прагматика.

В этом разделе мы обрисуем проблемы, связанные с дублированием, и предложим общие стратегии по тому, как с ним справиться.

Как возникает дублирование?

Большинство наблюдаемых явлений дублирования подпадают под одну из следующих категорий:

• **Навязанное дублирование.** Разработчики чувствуют, что у них нет выбора – им кажется, что дублирования требует среда окружения.

• **Неумышленное дублирование.** Разработчики не осознают, что они тиражируют информацию.

• **Нетерпеливое дублирование.** Разработчики ленятся и осуществляют дублирование, потому что им кажется, что так проще.

• **Коллективное дублирование.** Фрагмент информации тиражируются несколькими членами одной команды разработчиков (или нескольких команд)

Рассмотрим эти четыре категории дублирования более подробно.

Навязанное дублирование

Иногда кажется, что нас заставляют осуществлять дублирование. Стандарты, по которым делается проект, могут потребовать наличия документов, содержащих дублированную информацию, или документов, которые тиражируют информацию в тексте программы. При наличии нескольких целевых платформ каждая из них требует отдельных языков программирования, библиотек и сред разработки, что заставляет нас тиражировать общедоступные определения и процедуры. Сами языки программирования требуют наличия ряда конструкций, которые тиражируют информацию. Все мы находились в ситуациях, когда были не в силах избежать дублирования. И все же зачастую находятся способы сохранения каждого фрагмента знания в одном и том же месте – в соответствии с принципом DRY – и облегчения нашей жизни одновременно. Вот некоторые методики:

Множественные представления информации. На уровне создания текста программы, нам часто необходимо представить одну и ту же информацию в различных формах. Предположим, мы пишем приложение «клиент-сервер» с использованием различных языков для клиента и сервера и должны представить некоторую общедоступную конструкцию и на первом, и на втором. Возможно, нам необходим класс, чьи атрибуты отражают схему таблицы базы данных. Может быть, вы пишете книгу и хотите включить в нее фрагменты программ, которые вы также хотели бы скомпилировать и протестировать.

Немного изобретательности – и дублирование вам не понадобится. Зачастую ответ сводится к написанию простого фильтра или генератора текста программы. Конструкции с использованием нескольких языков можно собрать из обычного представления метаданных, применяя простой генератор текста программ всякий раз при осуществлении сборки программы (пример этого показан на рисунке 3.4). Определения класса могут быть сгенерированы автоматически из интерактивной схемы базы данных или из метаданных, используемых для построения схемы изначально. Фрагменты программ в этой книге вставлялись препроцессором всякий раз при форматировании текста. Уловка состоит в том, чтобы сделать процесс активным:

это не может быть однократным преобразованием, в противном случае мы опять окажемся в положении людей, тиражирующих данные.

Документация в тексте программы. Программистов учат комментировать создаваемый ими текст программы: удачный текст программы снабжен большим количеством комментариев. К сожалению, им никогда не объясняли, зачем тексту программы нужны комментарии: неудачному тексту требуется большое количество комментариев.

Принцип DRY говорит о сохранении низкоуровневого знания в тексте программы, частью которого он является, и сохранении комментариев для других, высокоуровневых толкований. В противном случае мы тиражируем знание, и каждое изменение означает изменение и в тексте программы, и в комментариях. Комментарии неизбежно устаревают, а ненадежные комментарии хуже, чем их отсутствие вообще. (Более подробная информация о комментариях содержится в разделе "Все эти сочинения").

Документация и текст программы. Вы пишете документацию, затем создаете текст программы. Что-то меняется, и вы исправляете документацию и обновляете текст. И документация, и текст содержат представления одного и того же знания. И все мы знаем, что в суматохе, когда приближается контрольный срок, а важные заказчики высказывают требования, обновление документации стараются отложить.

Однажды Дэйв Хант работал над переключателем телекса на разные языки. Вполне понятно, что заказчик требовал исчерпывающей тестовой спецификации, а также того, чтобы программы проходили полное тестирование при поставке каждой новой версии. Чтобы убедиться в том, что тесты находились в точном соответствии со спецификацией, команда сгенерировала их автоматически из самого документа. Когда заказчик вносил исправления в спецификацию, автоматически изменялся и тестовый набор программ. Команда убедила заказчика, что, после того как процедура прошла нормально, генерация приемочных тестов длилась лишь несколько секунд.

Языковые аспекты. Многие языки навязывают значительное дублирование в исходном тексте программы. Зачастую это происходит, когда язык отделяет интерфейс модуля от его реализации. Языки C и C++ используют файлы заголовка, которые тиражируют имена и печатают информацию о переменных экспорта, функциях и классах (для C++). Язык Object Pascal даже тиражирует эту информацию в том же самом файле. Если вы используете удаленные вызовы процедур или технологию CORBA[URL 29], то при этом происходит дублирование интерфейсной информации в спецификации интерфейса и тексте программы, его реализующей.

Не существует простой методики, позволяющей преодолеть требования языка. В то время как некоторые среды разработки скрывают потребность в файлах заголовка, генерируя их автоматически, а язык Object Pascal позволяет вам сокращать повторяющиеся объявления функции, в общем случае вы используете то, что вам дано. По крайней мере, для большинства языковых аспектов, файл заголовка, который противоречит реализации, будет генерировать некоторое сообщение об ошибке компиляции или компоновки.

Также стоит подумать о комментариях в файлах заголовка и реализации. В дублировании комментария функции или заголовка класса в этих двух файлах нет абсолютно никакого смысла. Файлы заголовка используются для документирования аспектов интерфейса, а файлы реализации – для документирования некоторых подробностей, которых пользователи вашей программы знать не должны.

Неумышленное дублирование

Иногда дублирование происходит в результате ошибок в проекте.

Рассмотрим пример из области транспорта. Пусть аналитик установил, что, наряду с прочими атрибутами, грузовик имеет тип, номерной знак и водителя. Аналогично, маршрут доставки груза представляет собой сочетание маршрута, грузовика и водителя. Мы создаем программы для некоторых классов, основанных на этом представлении.

Но что происходит, если водитель по имени Салли заболевает и приходится менять водителя? Классы Truck и DeliveryRoute содержат описание водителя. Какой из них мы должны изменить? Ясно, что это дублирование неудачно. Нормализуйте его в соответствии с базовой бизнес-моделью – необходим грузовику водитель как часть базового набора атрибутов? А маршрут? Возможно, необходим третий объект, который связывает воедино водителя, грузовик и маршрут. Каким бы ни было окончательное решение, стоит избегать этого типа ненормализованных данных.

Есть не столь очевидный тип ненормализованных данных, который имеет место при наличии множественных взаимозависимых элементов данных. Рассмотрим класс, представляющий отрезок:

```
class Line {
public:
 Point start;
 Point end;
 double length:
};
```

На первый взгляд, этот класс может показаться разумным. Отрезок явно имеет начало и конец и всегда будет иметь длину (даже если она нулевая). Но происходит дублирование. Длина определяется начальной и конечной точками: при изменении одной из точек длина меняется. Лучше сделать длину вычисляемым полем:

```
class Line {
public:
 Point start;
 Point end;
 double length() {return start.distanceTo(end);}
};
```

Позже, в ходе разработки, вы можете нарушить принцип "Не повторяй самого себя" в силу требований к производительности. Зачастую это происходит, когда вам необходимо кэшировать данные во избежание повторения дорогостоящих операций. Эта уловка призвана ограничить воздействие. Нарушение принципа не подвержено воздействию внешнего мира: лишь методы в пределах класса должны поддерживаться в надлежащем состоянии.

```
class Line {
private:
 bool changed;
 double length;
 Point start;
 Point end;
public:
void setStart(Point p) {start = p; changed = true;}
void setEnd(Point p) {end = p; changed = true;}
Point getStart(void) {return start;}
Point getEnd(void) {return end;}
```

```
double getLength() {
  if (changed) {
    length = start.distanceTo(end);
    changed = false;
  }
return length;
}
};
```

Этот пример также иллюстрирует важный аспект для объектно-ориентированных языков типа Java и C++. Там, где это возможно, всегда используются функции средства доступа – для чтения и записи атрибутов объектов [6]. Это облегчает добавление функциональных возможностей (типа кэширования) в будущем.

Нетерпеливое дублирование

Каждый проект испытывает давление времени – силы, которая может двигать лучшими из нас, заставляя идти напролом. Вам нужна подпрограмма, подобная уже написанной вами? Вас соблазнит возможность копирования и внесения лишь нескольких изменений? Вам нужно значение, чтобы представить максимальное число точек? Если я изменю файл заголовка, целый проект должен быть перестроен. Может, мне просто использовать константы в этом месте?… и в этом… и в том… Нужен класс, подобный тому, который есть в системе поддержки Java? У вас в распоряжении имеется исходный текст, так почему бы просто его не скопировать и не внести необходимые изменения (несмотря на лицензионное соглашение)?

Если вы чувствуете, что поддаетесь искушению, вспомните банальный афоризм: "Тише едешь – дальше будешь". Экономя несколько секунд в данный момент, вы потенциально теряете целые часы. Подумайте об аспектах, относящихся к "проблеме 2000 года". Многие из них были вызваны ленью разработчиков, которые не сделали параметризацию размера полей даты (или не внедрили централизованные библиотеки служб доступа к дате).

Нетерпеливое дублирование легко обнаруживается и устраняется, но это требует дисциплины и желания потратить время в настоящий момент, чтобы избежать головной боли впоследствии.

Коллективное дублирование

Самый трудный в обнаружении и обработке тип дублирования – коллективный – возникает между различными разработчиками проекта. Целые наборы функциональных возможностей могут тиражироваться по неосторожности, и это дублирование может оставаться незамеченным на протяжении многих лет, что приводит к возникновению проблем при сопровождении. Нам известно, как в одном из штатов США компьютерные системы, установленные в правительственных учреждениях, проверялись на наличие "проблемы 2000 года". Аудиторы обнаружили свыше 10000 программ, каждая из которых по-своему осуществляла проверку правильности номера карточки социального страхования.

На высоком уровне с проблемой можно справиться при наличии ясного проектного решения, сильного технического руководителя проекта (см. 'Команды прагматиков") и разделения обязанностей в пределах проекта. Однако на уровне модуля проблема является более коварной. Обычно необходимые функциональные возможности или данные, не относящиеся к

очевидной области ответственности, могут реализовываться много раз.

Мы полагаем, что лучший способ справиться с этим – поощрять активное и частое взаимодействие между разработчиками. Устраивайте форумы для обсуждения общих проблем. (При работе над предыдущими проектами мы организовывали конференции в сети, чтобы позволить разработчикам обмениваться идеями и задавать вопросы. Этим обеспечивается ненавязчивый способ общения – даже на нескольких сайтах – при сохранении непрерывной хронологии всего высказанного). Назначьте одного из членов команды библиотекарем проекта, чьей обязанностью будет обеспечение обмена знаниями. Организуйте специальное место в каталоге с исходными текстами, в котором будут сохраняться сервисные подпрограммы и скрипты. Обратите особое внимание на чтение исходного текста и документации других членов команды, неформально или при анализе текста программы. При этом вы отнюдь не шпионите за ними – вы учитесь у них. И помните, что доступ к тексту программы осуществляется по взаимной договоренности – вас не должно коробить, если и другие члены команды сосредоточенно изучают (или вынюхивают?) ваш текст программы.

Подсказка 12: Сделайте так, чтобы программу можно было легко использовать повторно

Все, что вы пытаетесь делать, способствует развитию среды, где проще находить и многократно использовать существующий материал, чем создавать его самому. Но если это непросто, люди не станут это делать. И если вы будете не в состоянии многократно использовать этот материал, вы рискуете заняться дублированием знания.

Другие разделы, относящиеся к данной теме:

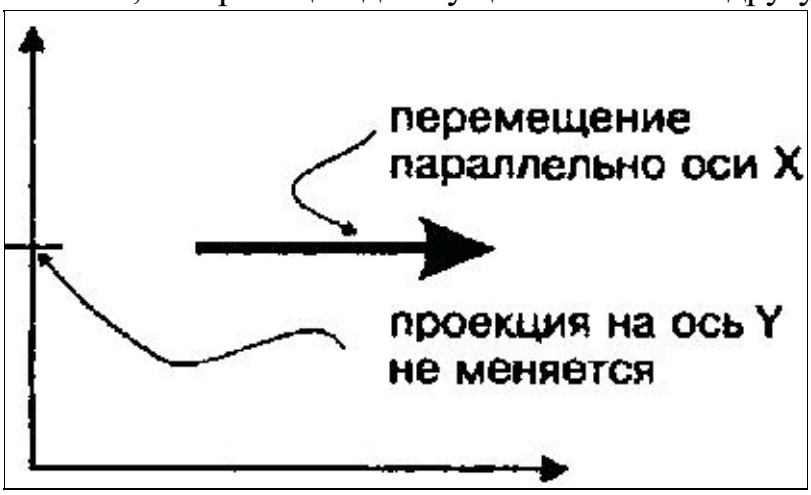
- Ортогональность
- Работа с текстом
- Генераторы исходных текстов
- Реорганизация
- Команды прагматиков
- Вездесущая автоматизация
- Все эти сочинения

8
Ортогональность

Ортогональность очень важна, если вы хотите создавать системы, которые легко поддаются проектированию, сборке, тестированию и расширению. Однако этому принципу редко обучают непосредственно. Часто он является лишь скрытым достоинством других разнообразных методик, которые вы изучаете. Это неправильно. Как только вы научитесь непосредственно применять принципы ортогональности, вы сразу заметите, как улучшилось качество создаваемых вами систем.

Что такое ортогональность?

Термин «ортогональность» заимствован из геометрии. Две линии являются ортогональными, если они пересекаются под прямым углом, например, оси координат на графике. В терминах векторной алгебры две такие линии являются независимыми. Если двигаться вдоль одной из линий, то проекция движущейся точки на другую линию не меняется.



Этот термин был введен в информатике для обозначения некой разновидности независимости или несвязанности. Два или более объекта ортогональны, если изменения, вносимые в один из них, не влияют на любой другой. В грамотно спроектированной системе программа базы данных будет ортогональной к интерфейсу пользователя: вы можете менять интерфейс пользователя без воздействия на базу данных и менять местами базы данных, не меняя интерфейса.

Перед тем как рассмотреть преимущества ортогональных систем, познакомимся с неортогональной системой.

Неортогональная система

Предположим, вы находитесь в экскурсионном вертолете, совершающем полет над Гранд-Каньоном, когда пилот, который совершил ошибку, наевшись рыбы за обедом, внезапно вскрикивает и теряет сознание. По счастливой случайности это происходит, когда вы парите на высоте 30 метров. Вы догадываетесь, что рычаг управления общим шагом несущего винта [7] обеспечивает подъем машины, так что, если его слегка опустить, вертолет начнет плавно снижаться. Однако когда вы пытаетесь сделать это, то осознаете, что жизнь – не такая уж простая штука. Вертолет клюет носом, и вас начинает вращать по спирали влево. Внезапно вы

понимаете, что управляете системой, в которой каждое воздействие имеет побочные эффекты. При нажатии на левый рычаг вам придется сделать уравновешивающее движение назад правым рычагом и нажать на правую педаль. Но при этом каждое из этих действий вновь повлияет на все органы управления. Неожиданно вам приходится жонглировать невероятно сложной системой, в которой любое изменение влияет на все остальные управляющие воздействия. Вы испытываете феноменальную нагрузку: ваши руки и ноги находятся в постоянном движении, пытаясь уравновесить все взаимодействующие силы.

Органы управления вертолетом определенно не являются ортогональными.

Преимущества ортогональности

Как показывает пример с вертолетом, неортогональные системы сложнее изменять и контролировать. Если составляющие системы отличаются высокой степенью взаимозависимости, то невозможно устранить какую-либо неисправность лишь на локальном уровне.

Подсказка 13: Исключайте взаимодействие между объектами, не относящимися друг к другу

Мы хотим спроектировать компоненты, которые являются самодостаточными: независимыми, с единственным, четким назначением; в книге Йордона и Константина [YC86] это явление называется сцеплением (cohesion). Когда компоненты изолированы друг от друга, вы уверены, что можно изменить один из них. не заботясь об остальных. Пока внешние интерфейсы этого компонента остаются неизменными, вы можете быть спокойны, что не создадите проблем, которые распространятся по всей системе.

С созданием ортогональных систем у вас появятся два больших преимущества: увеличение производительности и снижение риска.

Увеличение производительности

• Изменения в системе локализуются, поэтому периоды разработки и тестирования сократятся. Легче написать относительно небольшие, самодостаточные компоненты, чем один большой программный модуль. Простые компоненты могут быть спроектированы, запрограммированы, протестированы и затем забыты – не нужно непрерывно менять существующий текст по мере того, как к нему добавляются новые фрагменты.

• Ортогональный подход также способствует многократному использованию компонентов. Если компоненты имеют определенную, четкую сферу ответственности, они могут комбинироваться с новыми компонентами способами, которые не предполагались при их первоначальной реализации. Чем меньше связанность в системах, тем легче их перенастроить и провести их обратное проектирование.

• При комбинировании ортогональных компонентов происходит едва заметное увеличение производительности. Предположим, что один компонент способен осуществлять AJ, а второй – N различных операций. Если эти компоненты ортогональны и комбинируются, то в сумме они способны осуществить M x N различных операций. Но если два компонента не являются

ортогональными, то они будут перекрываться и результат их действия будет меньшим по сравнению с ортогональными компонентами. Вы получаете большее количество функциональных возможностей в пересчете на единичное усилие, если комбинируете между собой ортогональные компоненты.

Снижение риска

Ортогональный подход приводит к снижению уровня риска, присущего любой разработке.

• Ошибочные фрагменты текста программы изолируются. Если модуль содержит ошибку, то вероятность ее распространения на всю систему уменьшается. Кроме того, ошибочный фрагмент может быть извлечен и заменен новым (исправленным).

• Конечный продукт (система) становится менее хрупким. Проблемы, появляющиеся при внесении небольших изменений и устранении недочетов на определенном участке, не проходят дальше этого участка.

• Ортогональная система способствует повышению качества тестирования, поскольку облегчается проектирование и тестирование отдельных ее компонентов.

• Вы не будете слишком сильно привязаны к определенному субподрядчику, программному продукту или платформе, поскольку интерфейсы между компонентами, производимыми фирмами-субподрядчиками, не будут играть главенствующей роли в проекте.

Рассмотрим некоторые из способов, при помощи которых вы сможете внедрить принцип ортогональности в вашу работу.

Проектные группы

Приходилось ли вам замечать, насколько эффективно работают проектные команды, все члены которых знают, что делать, и полностью отдают себя делу, тогда как в других командах сотрудники постоянно препираются между собой и не собираются ни в чем уступать друг другу?

Зачастую это не что иное, как проблема ортогональности. Если команды организованы с большим числом перекрытий, то сотрудники путают свои должностные обязанности. Для любого изменения необходимо собирать всю команду, поскольку оно, может быть, затронет каждого.

Как разбить команду на группы с четкими обязанностями и минимальным перекрытием? На этот вопрос нет простого ответа. В некоторой степени это зависит от проекта и вашего анализа областей, которые в перспективе могут измениться. Это также зависит от людей, находящихся в вашем распоряжении. Мы предпочитаем отделять инфраструктуру от приложения. Каждому из основных инфраструктурных компонентов (база данных, интерфейс связи, промежуточное программное обеспечение и т. д.) приписывается только ему принадлежащая группа. Подобным образом производится и разделение функциональных возможностей приложения. После этого мы изучаем людей, которые имеются в нашем распоряжении на данный момент (или планируем их появление в будущем), и сообразно этому корректируем состав групп.

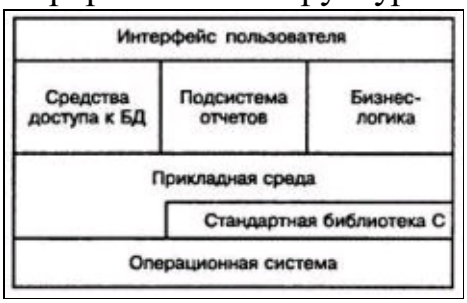
Вы можете неформально определить уровень ортогональности структуры проектной команды. Для этого просто посмотрите, скольких людей необходимо привлечь к обсуждению каждого изменения, требуемого со стороны. Чем больше эта цифра, тем ниже уровень ортогональности группы. Отсюда ясно, что ортогональная команда работает более эффективно. (Высказав это, мы тем самым поощряем стремление сотрудников более мелких подразделений

постоянно общаться друг с другом.)

Проектирование

Большинство разработчиков знакомо с потребностью в проектировании ортогональных систем, хотя они наверняка используют термины «модульный», "компонентно-ориентированный" и «многоуровневый» для описания конкретного процесса. Системы должны быть скомпонованы из набора взаимодействующих модулей, каждый из который реализует функциональные возможности независимо от других. Иногда эти компоненты объединены в уровни, каждый из которых обеспечивает некий уровень абстракции. Данный многоуровневый подход является мощным методом проектирования ортогональных систем. Поскольку на каждом уровне используются только абстракции, обеспеченные на низших уровнях, можно легко изменить основные реализации, не затрагивая самой программы. Иерархическое представление также уменьшает риск появления неконтролируемых зависимостей между модулями. Иерархическое представление часто показываются с помощью диаграмм, как на рисунке 2.1.

Рис. 2.1. Типичная диаграмма иерархической структуры



Существует простой тест на ортогональность проектирования. Как только вы составили схему компонентов, спросите себя: "Сколько модулей подвергнутся воздействию, если я резко изменю требования по конкретной функции?" В ортогональной системе ответ должен быть «один» [8]. Перемещение кнопки на панели графического интерфейса пользователя не должно требовать внесения изменений в схему базы данных. Добавление контекстно-зависимой справки не должно изменить подсистему выставления счетов.

Рассмотрим сложную систему контроля и управления отопительной установкой. Первоначально требовалось наличие графического интерфейса, но затем требования были изменены, с тем чтобы добавить систему речевого ответа и управления установкой при помощи телефона с тональным набором. В ортогонально спроектированной системе для этого вам пришлось бы изменить только модули, связанные с интерфейсом пользователя, а основная логика управления предприятием остается неизменной. На самом деле, если вы тщательно структурируете систему, то у вас должна быть возможность поддержки обоих интерфейсов при наличии одной и той же программной базы. В разделе "Всего лишь представление" говорится о написании программ, в которых отсутствует связанность, используя парадигму "модель-представление-контроллер" (Model-View-Controller), подходящую в данной ситуации.

Стоит спросить себя, как защитить вашу конструкцию от изменений в окружающем мире. Например, вы пользуетесь номером телефона в качестве идентификатора заказчика. Что произойдет, если телефонная станция изменит коды междугородной связи? Не полагайтесь на свойства предметов, которыми не можете управлять.

Инструментарии и библиотеки

Будьте внимательным, чтобы сохранить ортогональность вашей системы при введении

инструментариев и библиотек, произведенных фирмами-субподрядчиками. Проявите мудрость при выборе технологии.

Однажды авторы работали над проектом, в котором требовалось, чтобы некий фрагмент программы на языке Java выполнялся автономно – на сервере и в удаленном режиме – на клиентской машине. В этом случае возможными вариантами распределения классов были технологии RMI и CORBA. Если удаленный доступ к классу обеспечивался при помощи RMI, то в этом случае каждое обращение к удаленному методу в этом классе могло бы привести к генерации исключения, означающей, что эта наивная реализация потребовала бы от нас обработки этого исключения всякий раз при использовании удаленных классов. В данном случае использование RMI явно не ортогонально: программа, обращающаяся к удаленным классам, не должна зависеть от их физического расположения. Альтернативный способ – технология CORBA – не налагает подобного ограничения: мы можем написать программу, для которой не имеет значения, где физически находятся классы.

Когда вы используете инструментарий (или даже библиотеку, созданную другими разработчиками), вначале спросите себя, не заставит ли он внести в вашу программу изменения, которых там быть не должно. Если схема долговременного хранения объекта прозрачна, то она ортогональна. Если же при этом требуется создание объектов или обращение к ним каким-либо особым образом, то она неортогональна. Отделение этих подробностей от вашей программы дает дополнительное преимущество, связанное с возможностью смены субподрядчиков в будущем.

Интересным примером ортогональности является система Enterprise Java Beans (EJB). В большинстве диалоговых систем обработки запросов прикладная программа должна обозначать начало и окончание каждой транзакции. В системе EJB эта информация выражена описательно в виде метаданных вне любых программ. Та же самая прикладная программа может работать в различных транзакционных средах EJB без каких-либо изменений. Вероятно, это станет прообразом многих операционных сред будущего.

Другой интересной проверкой на ортогональность является технология Aspect-Oriented Programming (AOP) – исследовательский проект фирмы Xerox Pare ([KLM+97] и [URL 49]). Технология AOP позволяет выразить в одном-единственном месте линию поведения, которая в противном случае была бы распределена по всему исходному тексту программы. Например, журнальные сообщения обычно генерируются путем явных обращений к некоторой функции записи в журнал по всему исходному тексту. Используя технологию AOP, вы реализуете процедуру записи в журнал ортогонально к записываемым данным. Используя версию AOP для языка Java можно записать сообщение журнала при входе в любой метод класса Fred, запрограммировав аспект:

```
aspect Trace {
 advise * Fred.*(…) {
   static before {
     Log.write("-» Entering " + thisJoinPoint.methodName);
   }
  }
}
```

При вплетении этого аспекта в текст вашей программы будут генерироваться трассировочные сообщения. Если этого не сделать, не будет и сообщений. В обоих случаях исходный текст остается неизменным.

Написание текста программы

Всякий раз, когда вы пишете программу, вы подвергаетесь риску снижения уровня ортогональности вашего приложения. Если вы постоянно не отслеживаете не только то, что вы делаете, но и весь контекст приложения, то существует опасность неумышленного дублирования функциональных возможностей в некотором другом модуле или выражения существующих знаний дважды.

Есть ряд методик, которые можно использовать для поддержки ортогональности:

• **Сохраните вашу программу «несвязанной».** Напишите «скромную» программу – модули, которые не раскрывают ничего лишнего для других модулей и не полагаются на их внедрение. Попробуйте применить закон Деметера [LH89], который обсуждается в разделе "Несвязанность и закон Деметера". При необходимости изменения состояния объекта это должен делать сам объект. В таком случае программа остается изолированной от реализации другой программы, а вероятность того, что система останется ортогональной, увеличивается.

• **Избегайте глобальных данных.** Всякий раз, когда ваша программа ссылается на глобальные данные, она привязывается к другим компонентам, использующим эти данные. Даже глобальные переменные, которые вы собираетесь использовать только для чтения, могут вызвать проблемы (например, если вам необходимо срочно изменить программу, сделав ее многопоточной). Вообще программа станет проще в понимании и сопровождении, если вы явно перешлете любой требуемый контекст в ваши модули. В объектно-ориентированных приложениях контекст часто пересылается как параметр к конструкторам объектов. В другой программе вы можете создать конструкции, содержащие контекст, и обходить ссылки на них.

Шаблон Singleton, упомянутый в книге "Design Patterns" [GHJV95], представляет собой способ подтвердить существование единственного представителя объекта определенного класса. Многие используют эти объекты типа Singleton как своего рода глобальную переменную (особенно при работе с языками типа Java, которые иначе не поддерживают технологию глобальных переменных). Будьте внимательны с шаблонами Singleton – они также могут приводить к ненужному связыванию.

• **Подобные функции.** Зачастую вы сталкиваетесь с набором функций, похожих друг на друга; возможно, они используют общий фрагмент в начале и конце программы, но в ее середине каждая пользуется своим алгоритмом. Дублированная программа является признаком структурных проблем. Для того чтобы составить программу лучше, следует обратить внимание на шаблон Strategy в книге "Design Patterns".

Пусть постоянное критическое отношение к вашей программе войдет у вас в привычку. Ищите любые возможности реорганизации для усовершенствования ее конструкции и повышения уровня ортогональности. Этот процесс называется реорганизацией, и он важен настолько, что в книге ему посвящен целый раздел (см. "Реорганизация").

Тестирование

Систему, спроектированную и реализованную ортогональным образом, намного проще тестировать. Поскольку взаимодействие между компонентами системы формализовано и ограничено, большая часть тестирования может осуществляться на уровне отдельных модулей. Это хорошо, поскольку подобное тестирование значительно легче поддается спецификации и выполнению, чем интеграционное тестирование. Мы предлагаем, чтобы каждый модуль был снабжен своим собственным встроенным тестом и эти тесты выполнялись автоматически как часть обычной процедуры сборки (см. "Программа, которую легко тестировать").

Процедура сборки модульного теста сама по себе является интересным тестом на

ортогональность. Что требуется, чтобы собрать и скомпоновать тест модуля? Должны ли вы задействовать большую часть системы только для того, чтобы скомпилировать или скомпоновать тест? В этом случае модуль очень хорошо связан с оставшейся частью системы.

Момент устранения ошибки также подходит для оценки ортогональности системы в целом. Когда вы сталкиваетесь с проблемой, оцените, насколько локален процесс ее устранения. Нужно изменить лишь один модуль, или изменения должны происходить по всей системе? Когда вы меняете что-либо, устраняются ли при этом все ошибки или происходит загадочное появление новых? Это удачный момент для внедрения автоматизации. Если вы применяете систему управления исходным текстом (что вы будете делать, прочитав раздел "Средства управления исходным текстом"), комментируйте устранение ошибок, когда вы осуществляете возвращение измененного модуля в библиотеку после тестирования. Затем вы можете генерировать ежемесячные отчеты, где анализируются тенденции в ряде исходных файлов, в которых производилось устранение ошибок.

Документация

Что удивительно, ортогональность применима и к документации. Координатами являются содержание и представление. Если документация действительно ортогональна, вы можете существенно изменить внешний вид, не изменяя содержания. Современные текстовые процессоры содержат стили и макрокоманды, которые помогают в этом (см. "Все эти сочинения").

Жизнь в условиях ортогональности

Ортогональность тесно связана с принципом DRY ("Не повторяй самого себя"). Используя этот принцип, можно свести к минимуму дублирование в пределах системы, а при помощи ортогональности уменьшить взаимозависимость между компонентами системы. Звучит неуклюже, но если вы используете принцип ортогональности в тесной связи с принципом DRY, вы обнаружите, что разрабатываемые вами системы становятся более гибкими, более понятными и более простыми в отладке, тестировании и сопровождении.

Когда вы присоединяетесь к проекту, в котором люди ведут отчаянную борьбу за внесение изменений, а каждое изменение приводит к появлению четырех новых проблем, вспомните кошмар с вертолетом. Вероятно, проект сконструирован и запрограммирован неортогонально. Пришло время реорганизации.

Другие разделы, относящиеся к данной теме:

- Пороки дублирования
- Средства управления исходным текстом
- Проектирование по контракту
- Несвязанность и закон Деметера
- Метапрограммирование
- Всего лишь представление
- Реорганизация
- Программа, которую легко тестировать

• Злые волшебники
• Команды прагматиков
• Все эти сочинения

Вопросы для обсуждения

• Рассмотрим различие между большими инструментальными средствами, ориентированными на графический интерфейс, которые обычно присутствуют в системах в среде Windows, и небольшими, но сочетаемыми между собой утилитами, работающими в режиме командной строки и присутствующими в командных оболочках. Какой набор является более ортогональным и почему? Какой из них легче использовать именно для той цели, для которой он предназначен? Какой из них легче скомбинировать с другими инструментальными средствами для решения вновь возникших проблемных вопросов?

• Язык C++ поддерживает множественное наследование, а язык Java позволяет классу реализовывать множественные интерфейсы. Как влияет на ортогональность использование этих средств? Есть ли различие в воздействии, которое оказывается в ходе использования множественного наследования и множественных интерфейсов? Есть ли разница в применении делегирования и наследования?

Упражнения

1. Создается класс Split, который расщепляет вводимые строки на поля. Какая из двух указанных ниже сигнатур класса Java имеет более ортогональную конструкцию? (Ответ см. в Приложении B.)

```
class Split 1 {
public Splitl(InputStreamReader rdr) {…
public void readNextLine() throws IOException {…
public int numFields() {…
public String getField(int fieldNo) {…
}

class Split2 {
public Split2(String line) {…
public int numFields() {…
public String getField(int fieldNo) {…
}
```

2. Какая конструкция обладает большей ортогональностью: немодальные или модальные диалоговые окна? (Ответ см. в Приложении B.)

3. Сравним процедурные языки и объектно-ориентированные технологии. Что дает более ортогональную систему? (Ответ см. в Приложении B.)

9
Обратимость

Нет ничего опаснее идеи, если это единственное, что у вас есть.

Эмиль-Огюст Шартье, Разговор о религии, 1938

Технические специалисты предпочитают простые и однозначные решения задач. Математические тесты, позволяющие с большой уверенностью сказать, что x = 2, намного лучше, чем нечеткие, но страстные очерки о миллионах причин Французской революции. К техническим специалистам присоединяются и менеджеры: однозначные и несложные ответы хорошо вписываются в электронные таблицы и проектные планы.

Если бы это находило отклик в реальном мире! К сожалению, сегодня икс может быть равен двум, а завтра он должен быть равен пяти, а на следующей неделе – трем. Ничто не вечно, и если вы всерьез полагаетесь на некоторое явление, то этим вы практически гарантируете, что оно непременно изменится.

Для реализации чего-либо всегда существуют не один-единственный способ и не одна фирма-субподрядчик. Если вы начинаете работать над проектом, недальновидно полагая, что для его осуществления имеется один-единственный способ, то вы можете быть неприятно удивлены. Многим проектным командам открывают глаза принудительно, по мере развития событий:

> "Но вы же сказали, чтобы мы использовали базу данных XYZI. Мы написали 85 % текста проекта – мы не можем изменить его в данный момент", – протестует программист. "Очень жаль, но наша фирма решила вместо нее взять за основу базу PDQ – для всех проектов. Это немое решение. Мы все должны переписывать тексты программ… Всем вам придется работать и по выходным – до особого распоряжения".

Конечно, принимаемые меры не должны быть столь драконовскими, сколь и неотложными. Но поскольку время идет, а ваш проект продвигается, вы можете оказаться в шатком положении. С принятием каждого важного решения проектная команда ставит перед собой все более узкую цель – ограниченную версию действительности, в которой имеется меньшее число вариантов.

К тому времени, когда многие важные решения уже приняты, цель уменьшится настолько, что, если она двинется с места или ветер изменит направление, или же бабочка в Токио взмахнет своими крылышками, вы промахнетесь [9]. И здорово промахнетесь.

Проблема состоит в том, что непросто дать задний ход важным решениям.

Как только вы решите использовать базу данных этой фирмы или архитектурный шаблон, или определенную модель развертывания (например, «клиент-сервер» вместо автономной модели), то вы становитесь на путь, с которого невозможно свернуть – лишь ценой огромных затрат.

Обратимость

Многие из тем, затронутых в данной книге, нацелены на создание гибкого, легко адаптируемого программного обеспечения. Следуя их рекомендациям – в особенности

принципу DRY, принципу несвязанности и использованию метаданных (см. ниже), нет нужды в принятии многих важных необратимых решений. Это и хорошо, поскольку вначале мы не всегда принимаем наилучшие решения. Мы придерживаемся некоторой технологии лишь для того, чтобы в один прекрасный день обнаружить, что не в состоянии нанять достаточное количество людей, обладающих необходимыми навыками. Стоит нам остановить свой выбор на некоторой фирме-субподрядчике, как ее сразу перекупают конкуренты. Требования, пользователи и аппаратные средства изменяются быстрее, чем мы разрабатываем программное обеспечение.

Предположим, что в начале проекта вы решили использовать реляционную базу данных, производимую фирмой А. Позже, во время нагрузочного тестирования, вы обнаруживаете, что база данных слишком медленная, а объектная база данных фирмы В работает быстрее. В большинстве случаев, вам не везет. Большую часть времени обращения к программам фирм-субподрядчиков запутываются в тексте программ. Но если вы действительно вычленили идею базы, поместив ее снаружи – в точку, где она просто обеспечивает сохранение состояния объектов (как служба), тогда вы обладаете достаточной гибкостью, чтобы менять коней на переправе.

Предположим, что проект начинается по модели «клиент-сервер», но затем, когда карты уже сданы, отдел маркетинга решает, что для некоторых заказчиков серверы слишком дороги и они хотят сделать автономную версию. Насколько сложным будет для вас этот переход? Поскольку речь идет о развертывании, для этого потребуется минимум несколько дней. Если бы времени требовалось больше, вы бы и не думали об обратимости. Обратная задача еще интереснее. Что будет, если возникнет необходимость в развертывании автономной версии разрабатываемого вами проекта по схеме «клиент-сервер» или по n-звенной модели? Это также не должно представлять затруднений.

Ошибка состоит в предположении, что любое решение высечено на камне, и в неготовности к случайностям, которые могут возникнуть. Вместо того, чтобы высекать решения на камне, рассматривайте их так, как будто они начерчены на морском песке. В любой момент может накатиться большая волна и смыть их.

Подсказка 14: Не существует окончательных решений

Гибкая архитектура

В то время как многие люди пытаются сохранить свои программы гибкими, вам также стоит подумать о том, чтобы обеспечить гибкость архитектуры, развертывания и интеграции продуктов фирм-субподрядчиков.

Технологии, подобные CORBA, могут помочь в защите компонентов проекта от изменений, происходящих в языке, на котором ведется разработка, или в платформе. Вдруг производительность Java на этой платформе не соответствует ожиданиям? Еще раз напишите программу клиента на языке C++, и больше ничего менять не нужно. Подсистема правил в C++ не отличается достаточной гибкостью? Перейдите к версии на языке Smalltalk. При работе с архитектурой CORBA вы должны обращать внимание только на заменяемый компонент, другие компоненты трогать не нужно.

Вы разрабатываете программы для Unix? Какой версии? Вы рассмотрели все из аспектов переносимости? Вы пишете для конкретной версии Windows? Какой – 3.1, 95, 98, NT, CE или же

2000? Насколько сложно будет обеспечить поддержку других версий? Если ваши решения характеризуются мягкостью и пластичностью, то это будет совсем несложно. Но это будет невозможно, если пакет неудачно сформирован, есть высокий уровень связанности, а в тексты программ встроена логика или параметры.

Вы не знаете точно, как отдел маркетинга собирается развертывать систему? Подумайте об этом заранее, и вы сможете обеспечить поддержку автономной модели, модели "клиент – сервер" или n-звенной модели только за счет изменений в файле конфигурации. Мы создавали программы, которые действуют подобным образом.

Обычно вы можете просто скрыть продукт фирмы-субподрядчика за четким, абстрактным интерфейсом. На самом деле мы могли это сделать с любым проектом, над которым мы работали. Но предположим, что вы не смогли изолировать его достаточно четко. Вам пришлось раскидать некоторые инструкции по всей программе? Поместите это требование в метаданные и воспользуйтесь автоматическим механизмом, наподобие Aspect (см. "Инструментарии и библиотеки") или Perl для вставки необходимых инструкций в саму программу. Какой бы механизм вы ни использовали, сделайте его обратимым. Если что-то добавляется автоматически, то оно может и удаляться автоматически.

Никто не знает, что может произойти в будущем, в особенности мы! Дайте вашей программе работать в ритме рок-н-ролла: когда можно – качаться, а когда нужно – энергично крутиться.

Другие разделы, относящиеся к данной теме:

- Несвязанность и закон Деметера
- Метапрограммирование
- Всего лишь представление

Вопросы для обсуждения

• Немного квантовой механики – пример с кошкой Шрёдингера. Предположим, что в закрытом ящике сидит кошка, и в нем же находится радиоактивная частица. Вероятность распада частицы на две равна 50 %. Если распад произойдет, кошка умрет. Если не произойдет, кошка останется жива. Итак, умирает кошка или остается жива? Согласно Шрёдингеру, верно и то, и другое. Всякий раз, когда происходит ядерная реакция, у которой имеются два возможных результата, происходит клонирование мира. В одном из двух миров данное событие произошло, а в другом – нет. Кошка жива в одном из миров и мертва в другом. Лишь открыв ящик, вы осознаете, в каком из миров находитесь вы.

Не удивительно, что программировать на перспективу так трудно.

Но подумайте об эволюции программы по аналогии с ящиком, в котором находится множество кошек Шрёдингера: каждое решение приводит к появлению иной версии будущего. Сколько сценариев будущего поддерживает ваша программа? Какие из них наиболее вероятны? Насколько сложно будет поддерживать их в определенный момент в будущем?

Хватит ли у вас смелости открыть ящик?

10
Стрельба трассирующими

На изготовку, по цели – пли!

Существует два способа стрельбы из пулемета в темное время суток [10]. Вы можете выяснить точно, где находится ваша цель (расстояние, высота и азимут). Вы можете определить погодные условия (температура, влажность, давление, направление ветра и так далее). Вы можете точно определить характеристики используемых вами патронов и пуль и их взаимодействие с реальным пулеметом, из которого вы стреляете. Затем вы можете воспользоваться таблицами или компьютером для вычисления точного азимута и угла возвышения ствола пулемета. Если все работает в точном соответствии с характеристиками, таблицы корректны, а погодные условия не меняются, то пули должны лечь близко к цели. Можно также использовать трассирующие пули.

Трассирующие пули помещаются на пулеметную ленту через равные промежутки наряду с обычными боеприпасами. При стрельбе фосфор, содержащийся в них, загорается и оставляет пиротехнический след, идущий от пулемета до любого места, в которое эти пули попадают. Если в цель попадают трассирующие пули, то, значит, в нее попадут и обычные.

Не удивительно, что стрельбу трассирующими предпочитают математическим расчетам. Обратная связь возникает немедленно, и поскольку трассирующие пули работают в той же среде, что и обычные боеприпасы, то внешние воздействия сведены к минимуму.

Возможно это слишком сильная аналогия, но она применима к новым проектам, особенно когда вы создаете то, чего раньше не было. Подобно стрелкам, вы пытаетесь поразить цель в темноте. Ваши пользователи никогда ранее не видели ничего подобного, поэтому их требования могут быть расплывчатыми. Вы же, в свою очередь, наверняка применяете алгоритмы, методики, языки или библиотеки, с которыми не знакомы, то есть сталкиваетесь с большим количеством неизвестных. И поскольку для выполнения проекта требуется время, вы можете с уверенностью гарантировать, что к моменту окончания работы среда, в которой вы работаете, изменится.

Классический способ решения проблемы – предельно специфицировать систему. Написать горы бумажной документации, регламентирующих каждое требование, связывая каждое неизвестное и ограничивая рабочую среду. Стрелять при помощи жесткого расчета. Один большой предварительный расчет, затем стрельнуть и надеяться.

Однако программисты-прагматики предпочитают стрелять трассирующими.

Программа, которую видно в темноте

Стрельба трассирующими пулями эффективна, поскольку эти пули работают в той же самой среде и подвержены тем же ограничениям, что и реальные пули. Они быстро оказываются у цели, так что стрелок получает немедленную обратную связь. И с практической точки зрения они представляют собой относительно экономичное решение.

Чтобы добиться того же эффекта в программах, мы ищем нечто такое, что позволяет нам быстро, наглядно и многократно проходить путь от требования до некоторой характеристики окончательной версии системы.

Подсказка 15: Пользуйтесь трассирующими пулями, для того чтобы найти цель

Однажды мы работали над сложным маркетинговым проектом с базой данных «клиент-сервер». Частью требований была способность определять и выполнять промежуточные запросы. Серверами являлся ряд реляционных и специализированных баз данных. Клиентский графический интерфейс пользователя, написанный на языке Object Pascal, использовал набор библиотек C для обеспечения интерфейса с серверами. Запрос пользователя хранился на сервере с использованием системы обозначений, подобной Lisp, до момента преобразования в оптимизированный SQL-запрос, предшествующего его выполнению. При этом возникло много неизвестных и много различных сред, и никто не знал наверняка, как же поведет себя графический интерфейс пользователя.

Это был отличный повод для применения программы трассировки. Мы разработали «скелет» внешнего интерфейса, библиотеки для представления запросов и конструкцию для преобразования сохраненного запроса в запрос, определенный базой данных. Затем мы свели все воедино и проверили, работает ли это. Все, что мы могли сделать в первоначальном варианте, был запрос, который выдавал перечень всех строк в таблице, но он доказал, что интерфейс пользователя мог взаимодействовать с библиотеками, библиотеки могли преобразовать запрос в последовательную и параллельную форму, а из результата сервер мог сгенерировать SQL-запрос. На протяжении следующих месяцев мы постепенно разрабатывали основную конструкцию, добавляя новую функциональную возможность путем параллельного наращивания каждого компонента программы трассировки. Когда интерфейс пользователя добавлял новый тип запроса, библиотека увеличивалась, и генерация SQL-запроса становилась более утонченной.

Программа трассировки не является одноразовой: вы пишете ее, чтобы сохранить. Она содержит всю проверку ошибок, структурирование, документацию и самоконтроль, которые имеются в любом фрагменте рабочей программы. Она просто не обладает всеми функциональными возможностями. Однако, как только вы добились сквозного соединения между компонентами вашей системы, вы можете проверить, насколько близко вы находитесь к цели, и в случае необходимости сделать поправку. Как только вы попали в цель, добавление функциональных возможностей облегчается.

Разработка программ трассировки находится в согласии с той идеей, что проект никогда не кончается: всегда будет потребность в изменениях и добавлении функций. Это – инкрементальный подход.

Обычная альтернатива является своего рода тяжеловесным техническим подходом: программа разделяется на модули, которые программируются в вакууме. Модули объединены в подсистемы, которые затем подлежат дальнейшему объединению, пока в один прекрасный день вы не получаете завершенное приложение. И только тогда приложение в целом может быть представлено пользователю и протестировано. Технология программы трассировки имеет много преимуществ:

• **Пользователи могут увидеть нечто, работающее еще до выпуска окончательной версии.** Если вам удалось передать суть делаемого вами (см. "Большие надежды"), то ваши пользователи будут осознавать, что видят перед собой еще нечто незрелое. Они не будут разочарованы отсутствием функциональных возможностей; будут гореть желанием увидеть некий видимый прогрессе создании их системы. По мере того как проект будет продвигаться, они начнут делать вложения. Эти пользователи и станут теми людьми, которые скажут вам о том, насколько близко к цели находится та или иная итерация.

• **Разработчики выстраивают некую структуру, в которой они работают.** Наибольший

страх вызывает лист бумаги, на котором ничего не написано. Если вы разработали все механизмы взаимодействия между модулями вашего приложения и воплотили их в тексте программы, то вашей команде не придется многое высасывать из пальца. Это делает труд каждого члена команды более производительным и способствует последовательности в их работе.

• **У вас есть платформа для интеграции.** Как только все компоненты системы связаны друг с другом, появляется некая среда, в которую можно добавлять новые фрагменты программ, прошедшие модульное тестирование. Вы будете заниматься интеграцией каждый день (иногда несколько раз вдень), не пытаясь интегрировать все сразу по методу "большого скачка". Воздействие каждого вновь вносимого изменения становится более очевидным, взаимодействия более ограниченными, поэтому отладка и тестирование будут более быстрыми и точными.

• **У вас есть что продемонстрировать.** Спонсоры проекта и руководство стремятся увидеть демонстрационные версии в самое неподходящее время. При наличии программы трассировки у вас всегда будет то, что можно им продемонстрировать.

• **Вы лучше ощущаете прогресс.** При разработке программы трассировки программисты работают над сценариями использования системы по очереди. Они заканчивают один сценарий и переходят к следующему. При этом гораздо проще определить производительность и продемонстрировать пользователю продвижение проекта. Поскольку каждая индивидуальная разработка меньше по объему, вы избежите создания монолитных программных блоков, о которых каждую неделю сообщается, что они готовы на 95 %.

При стрельбе трассирующими вы не всегда попадаете в цель

Трассирующие пули показывают, что вы куда-то попали. Это не обязательно должна быть ваша цель. Затем вы корректируете прицел, пока пули не попадают в цель. В этом-то все и дело.

То же самое относится и к программе трассировки. Вы используете методику в ситуациях, когда не уверены на 100 %, куда же вам двигаться. Не стоит удивляться, если две первых попытки сорвались: пользователь говорит: "Это совсем не то, что я имел в виду", нужные данные становятся недоступными в самый неподходящий мо-мент, и явно возникают проблемы с производительностью. Выработайте подход для изменения того, что мешает приблизиться к цели, и будьте благодарны судьбе, что вы используете скудную методологию разработки. Небольшой фрагмент программы отличается малой инерцией – его легко и быстро изменить. Вы сможете собрать отклики на ваше приложение и сгенерировать новую, более точную версию быстрее и дешевле. И поскольку каждый основной компонент приложения представлен в программе трассировки, ваши пользователи могут быть уверены – то, что они видят, основано на реальности, а не на бумажных спецификациях.

Программа трассировки и создание прототипов

Вы могли бы подумать, что принцип программы трассировки – это то же самое, что и чем создание прототипов, только с более агрессивным названием. Отличие есть. Цель работы с прототипом – исследование определенных характеристик (аспектов) конечной версии системы. Создавая истинный прототип, вы отбросите все то, что критиковали при опробовании принципа, и перепишете его надлежащим образом, используя полученные уроки.

Например, вы создаете приложение, которое помогает транспортным компаниям определять, как упаковывать ящики нестандартного размера в контейнеры. Помимо всего

прочего, интерфейс пользователя должен быть интуитивно понятным, а алгоритмы, используемые для определения оптимальной упаковки, очень сложны.

Вы могли бы создать интерфейс для конечных пользователей при помощи соответствующих инструментальных средств. Вашей программы достаточно для того, чтобы сделать интерфейс восприимчивым к действиям пользователя. Как только пользователи согласятся с компоновкой интерфейса, вы можете отбросить его и переписать на этот раз на основе бизнес-логики, используя целевой язык. Аналогично, вы можете захотеть создать прототип ряда алгоритмов, которые осуществляют реальную упаковку. Вы можете запрограммировать функциональные тесты на высокоуровневом, «всепрощающем» языке типа Perl и затем запрограммировать низкоуровневые тесты производительности на языке, который более близок к машинному. В любом случае, как только вы приняли решение, необходимо начать сначала и запрограммировать алгоритмы в окончательной версии среды, которая взаимодействует с внешним миром. Это и есть создание прототипов, и это очень полезно.

Подход типа "стрельба трассирующими" обращается к иной проблеме. Вам необходимо знать, как работает приложение в целом. Вы хотите показать вашим пользователям, как на практике осуществляется взаимодействие, и дать им «скелет» архитектуры, на который наращивается тело программы. В этом случае вы можете сконструировать программу трассировки, состоящую из тривиальной реализации алгоритма упаковки контейнера (возможно, нечто вроде FIFO), и простой, но работающий интерфейс пользователя. Как только вы соедините все компоненты приложения, у вас уже есть каркас, который можно представить вашим пользователям и разработчикам. Спустя некоторое время вы добавляете к этому каркасу новую функциональную возможность, заменяя заглушки программами. Но сам остов остается нетронутым, и вы знаете, что система будет вести себя так же, как и в тот момент, когда завершалась первая программа трассировки.

Различие достаточно важно, чтобы гарантировать повторяемость. Прототипы генерируют одноразовую программу. Программа трассировки является скудной, но завершенной; она образует часть «скелета» конечной версии системы. Рассматривайте создание прототипов как рекогносцировку и сбор данных разведки до начала стрельбы трассирующими.

Другие разделы, относящиеся к данной теме:

- Приемлемые программы
- Прототипы и памятные записки
- Западня спецификации
- Большие надежды

11
Прототипы и памятные записки

Для опробования определенных идей во многих отраслях промышленности используются прототипы; это дешевле, чем организовывать полномасштабное производство. Например, в автомобильной промышленности для новой модели автомобиля может быть построено несколько различных прототипов. Каждый из них конструируется для проверки определенных свойств автомобиля – аэродинамики, дизайна, свойств конструкции и т. д. Для испытания в аэродинамической трубе изготовляется модель из глины, для отдела дизайна создается модель из бальзовой древесины и клейкой ленты и т. д. Некоторые автомобильные фирмы идут дальше и осуществляют значительную часть работы по моделированию при помощи компьютеров, что приводит к еще большему сокращению расходов. В этом случае нет необходимости в реальном изготовлении рискованных элементов конструкции для их опробования.

Мы создаем прототипы программ тем же образом и по тем же причинам – для анализа и выявления риска, предлагая возможности для коррекции при существенно меньших затратах. Подобно тому, как это делается в автомобильной промышленности, мы можем использовать прототип для опробования одного или нескольких характеристик проекта.

Мы склонны полагать, что основой прототипов являются программы, но не это не всегда так. Подобно тому, как это делается в автомобильной промышленности, мы можем строить прототипы из различных материалов. Памятные записки великолепно подходят для создания прототипов таких динамических объектов, как логика документооборота и прикладная логика. Прототип интерфейса может моделироваться на лекционной доске, как модель без функциональных возможностей, изображенная с помощью графической программы или программы-построителя интерфейса.

Прототипы разрабатываются для того, чтобы ответить лишь на несколько вопросов, поэтому их разработка намного дешевле и быстрее, чем приложения, которые идут в производство. Программа может игнорировать незначительные подробности – незначительные в данный момент, но позже могущие оказаться для пользователя очень важными. Если, к примеру, вы создаете прототип графического интерфейса пользователя, то можете смириться с неправильными результатами или данными. С другой стороны, если вы просто исследуете характеристики вычислений или производительности, можете обойтись скудным графическим интерфейсом пользователя или вообще без него.

Но если вы работаете в среде, где нельзя отказаться от подробностей, тогда необходимо спросить себя, а нужно ли вообще создавать прототип. Возможно, в этом случае лучше всего подходит стиль разработки типа "стрельба трассирующими" (см. "Стрельба трассирующими").

Для чего создаются прототипы

Какие объекты можно изучать при помощи прототипов? Все, что характеризуются наличием риска. Все, что не подвергались тестированию ранее или являются абсолютно критичными для конечного варианта системы. Вес, что является недоказанным, экспериментальным или сомнительным. Вес то, с чем вы еще не освоились. Вы можете создавать прототипы:

- Архитектуры
- Новой функциональной возможности уже существующей системы
- Структуры или содержания внешних данных

• Инструментальных средств или компонентов, выпущенных фирмами-субподрядчиками
• Рабочих характеристик
• Дизайна интерфейса пользователя

Создание прототипов способствует приобретению опыта. Значение этого опыта заключается не в созданной программе, а в полученных уроках. В этом и состоит смысл создания прототипов.

Подсказка 16: Создавайте прототипы, чтобы учиться на них

Как использовать прототипы

Какими деталями можно пренебречь при построении прототипа?

• **Корректность.** Там, где это приемлемо, вы сможете использовать фиктивные данные.

• **Завершенность.** Прототип может функционировать лишь в ограниченном смысле, возможно, лишь с одним наперед заданным фрагментом входных данных и одним пунктом меню.

• **Надежность.** Процедура проверки ошибок, вероятно, будет неполной или будет отсутствовать полностью. Если вы отклоняетесь от определенного пути, то прототип может выйти из строя и сгореть, как ракета. Это нормально.

• **Стиль.** Неприятно признавать это, но прототип программы не имеет большого значения для комментариев или документации. При работе с прототипом можно написать горы документации, но сравнительно малая ее часть будет посвящена собственно прототипу системы.

Поскольку в прототипе детали отодвигаются на второй план, а в центре рассмотрения оказываются определенные аспекты системы, вам может показаться реальным создание прототипов с использованием языка очень высокого уровня – выше уровня языка остальной части проекта (язык типа Perl, Python или Tel). Язык сценариев высокого уровня позволяет опускать многие детали (включая указание типов данных) и при этом создавать функциональный (хотя и неполный и медленный) фрагмент программы [11]. Если вам необходимо создать прототип интерфейсов пользователей, изучите инструментальные средства типа Tcl/Tk, Visual Basic, Powerbuilder или Delphi.

Языки сценариев хороши для использования в качестве «клея» при соединении низкоуровневых фрагментов в новые сочетания. При работе в системе Windows язык Visual Basic может «скреплять» средства управления COM. В более общем смысле вы можете использовать языки типа Perl и Python для связывания воедино низкоуровневых библиотек языка C – вручную или автоматически при помощи инструментов наподобие бесплатного SWIG [URL 28]. Используя этот подход, вы можете быстро собрать существующие компоненты в новые конфигурации, чтобы посмотреть, как они работают.

Создание прототипов архитектуры

Многие прототипы создаются, чтобы смоделировать рассматриваемую систему в целом. В отличие от подхода типа "стрельба трассирующими", ни один из отдельных модулей в прототипе системы не должен быть особенно функциональным. На самом деле вам даже не нужно писать программу для создания прототипов – вы можете создать прототип на

лекционной доске, при помощи памятных записок или каталожных карточек. Вы пытаетесь понять то, как система выглядит в собранном виде, опуская детали. Вот некоторые из конкретных областей, которые вы можете обнаружить в архитектурном прототипе:

• Четко ли определены обязанности основных компонентов, и являются ли они приемлемыми?

• Четко ли определена совместная работа основных компонентов?

• Сведено ли к минимуму связывание?

• Можно ли идентифицировать потенциальные источники дублирования?

• Можно ли применить определения интерфейсов и ограничения?

• Обладает ли каждый из модулей путем доступа к данным, требуемым ему в ходе выполнения? Может ли он получить такой доступ в случае необходимости?

Последний пункт приносит большинство сюрпризов и наиболее ценных результатов, основанных на опыте создания прототипов.

Как не надо использовать прототипы

Перед тем как вы займетесь созданием любого прототипа, основанного на программе, убедитесь, что все понимают – вы пишете одноразовую программу. Прототипы могут быть обманчиво привлекательными для людей, которые не знают, что это всего лишь прототипы. Вы должны очень четко уяснить – эта программа одноразовая, незавершенная и не может быть завершена.

Легко впасть в заблуждение из-за очевидной завершенности демонстрационного прототипа, и спонсоры проекта или менеджмент могут настаивать на развертывании прототипа (или его потомства), если вы заранее не определите, что можно ожидать от прототипа. Напомните им, что вы, конечно, можете создать великолепный прототип новой модели автомобиля из бальзовой древесины и клейкой ленты, но вы же не поедете на нем в час пик!

Если вы полагаете, что в вашей среде или культуре существует большая вероятность того, что назначение прототипа программы может быть истолковано неправильно, вам лучше воспользоваться подходом "стрельба трассирующими". Вы получите некий жесткий каркас, на котором можно основывать будущие разработки.

При надлежащем использовании прототип может сэкономить вам огромное количество времени, денег, головной боли и мучений за счет идентификации и исправления потенциальных проблем в самом начале цикла разработки – затраты на устранение ошибок будут недорогими и не вызовут затруднений.

Другие разделы, относящиеся к данной теме:

• Мой исходный текст съел кот Мурзик

• Общайтесь!

• Стрельба трассирующими

• Большие надежды

Упражнения

4. Специалисты по маркетингу хотели бы сесть и вместе с вами провести мозговой штурм

по дизайну нескольких интернет-страниц. Они думают об активных картах ссылок – для перехода к другим страницам. Но они не могут определиться с моделью ссылки: это могут быть изображения автомобиля, телефона или дома. У вас имеется перечень целевых страниц и содержания; они хотели бы увидеть несколько прототипов. Да, кстати, в вашем распоряжении 15 мин. Какими инструментами вы могли бы воспользоваться? (Ответ см. в Приложении В.)

12
Языки, отражающие специфику предметной области

Границы моего языка есть границы моего мира.

Людвиг фон Витгенштейн

Языки программирования влияют на то, как вы думаете о проблеме, и на то, как вы думаете об общении. В каждом языке имеются свои особенности – ученые словечки типа "статический и динамический контроль типов", "раннее и позднее связывание", "модели наследования" (простое, множественное или отсутствие) – все они могут предложить определенные решения или затруднить их. Решение, создаваемое в стиле Lisp, отличается от решения, основанного на мышлении приверженца языка С, и наоборот. Верно и обратное (и по нашему мнению, более важное) – язык, отражающий специфику данной области, может, со своей стороны, предложить решение в области программирования.

Мы всегда пытаемся написать программу, используя словарь, характерный для прикладной области (см. "Ловушка требований", где предлагается использовать проектный глоссарий). В ряде случаев можно перейти на следующий уровень и действительно программировать, пользуясь словарем, синтаксисом и семантикой предметной области.

Пользователи предложенной системы должны быть в состоянии точно изложить, как она должна работать:

> Ожидать прихода сообщений, определенных нормативом 12.3 фирмы ABC, по каналам связи X.25, преобразовать их в формат 43B фирмы XYZ, ретранслировать на спутниковый канал связи и сохранить для анализа в будущем.

Если ваши пользователи располагают набором подобных четких инструкций, то вы можете изобрести мини-язык, скорректированный в соответствии с прикладной областью и выражающий именно то, что им нужно:

```
From X25LINE1 (Format=ABC123) {
  Put TELSTAR1 (Format=XYZ43B);
  Store DB;
}
```

Этот язык не должен быть исполняемым. В своем исходном виде он мог бы просто фиксировать требования пользователя – спецификации. Однако вы наверняка посчитали возможным пойти дальше и фактически реализовать язык. Ваша спецификация превратилась в исполняемую программу.

После того как вы написали приложение, пользователи предъявляют вам новое требование: сообщения с отрицательным балансом не должны сохраняться и должны отсылаться обратно по каналам связи X.25 в первоначальном формате:

```
From X25LINE1 (Format=ABC123) {
  if (ABC123.balance «0) {
    Put X25L1NE1 (Format=ABC123);
  }
  else {
    Put TELSTAR1 (format=XYZ43B);
```

```
    Store DB;
  }
 }
```

Несложно? При наличии надлежащей поддержки вы можете программировать значительно ближе к прикладной области. Мы не предлагаем, чтобы ваши конечные пользователи программировали на этих языках. Вместо этого вы даете самому себе инструмент, который позволяет вам работать ближе к их области.

Подсказка 17: Программируйте ближе к предметной области вашей задачи

Мы полагаем, что вам следует рассмотреть способы перемещения вашего объекта ближе к предметной области проблемы – будь то простейший язык для конфигурирования и управления прикладной программой или же более сложный язык для обозначения правил или процедур. При составлении программы на более высоком уровне абстракции вам легко сосредоточиться на решении проблем предметной области и вы можете проигнорировать мелкие детали, связанные с реализацией.

Помните, что с приложением работают многие пользователи. Существует конечный пользователь, который понимает правила предметной области и то, что должно быть на выходе программы. Есть также вторичные пользователи: обслуживающий персонал, менеджеры, занимающиеся конфигурированием и тестированием, программисты служб поддержки и сопровождения и будущие поколения разработчиков. У каждого из этих пользователей есть собственная предметная область, и для всех них вы можете генерировать мини-среды и языки.

Ошибки, отражающие специфику предметной области

Если вы работаете в определенной предметной области, то можете осуществить и проверку правильности данных, характерных для нее, сообщая о проблемах языком, понятным вашим пользователям. Рассмотрим программу коммутации каналов, приведенную выше. Предположим, что пользователь неправильно обозначил наименование формата:

From X25LINE1 (Format=AB123)

Если подобное происходит в универсальном языке программирования, то выдается стандартное сообщение об ошибке:

Syntax error: undeclared identifier

Используя мини-язык, вместо этого можно создать сообщение об ошибке, используя словарь предметной области:

"AB123" is not a format. Known formats are ABC123, XYZ43B, PDQB, and 42.

Реализация мини-языка

В самом простейшем варианте мини-язык может реализовываться в строчно-ориентированном, легко анализируемом формате. Практически мы используем эту форму больше, чем любую другую. Ее просто проанализировать при помощи инструкций switch или используя регулярные выражения в языках сценариев типа Perl. Ответ к упражнению 5

(Приложение B) показывает простую реализацию мини-языка на языке C.

Вы можете реализовать и более сложный язык с более формальным синтаксисом. Хитрость состоит в том, чтобы вначале определить синтаксис, используя систему обозначений, подобную нормальной форме Бэкуса-Наура [12]. Как только вы определили грамматику, ее обычно легко преобразовать во входной синтаксис для генератора. Программисты, работающие с языками C и C++, давно используют yacc (или его бесплатную версию bison [URL 27]). Подробное описание этих программ приводится в книге "Lex and Yacc" [LMB92]. Программисты, работающие с языком Java, могут поработать с программой javaCC, которая находится на сайте [URL 26]. В ответе к упражнению 7 (Приложение B) дана программа грамматического разбора, написанная с помощью bison. Пример показывает, что как только вы изучите синтаксис, написание мини-языков не представит сложности.

Существует другой способ реализации мини-языка: расширить существующий. Например, можно интегрировать функциональные возможности на уровне приложения при помощи Python [URL 9] и написать нечто вроде [13]:

```
record = X25UNE1.get(format=ABC123)
if (record.balanc<0):
  X25UNE1.put(record, format=ABC123)
else:
  TELSTR1.put(record, format=XYZ43B)
  DB.store(record)
```

Языки управления данными и процедурные языки

Реализуемые вами языки могут использоваться двумя различными способами.

Языки управления данными создают некую форму структуры данных, используемой приложением. Эти языки часто используются для представления конфигурации.

Например, программа sendmail применяется во всем мире для маршрутизации электронной почты в сети Интернет. Она обладает многими достоинствами и преимуществами, которые управляются огромным файлом конфигурации, написанном на собственном языке конфигурирования программы sendmail:

```
Mlocal, P=/usr/bin/procmail,
F=lsDFMAw5:/|@qSPfhn9,
S=10/30, R=20/40,
T=DNS/RFC822/X-Unix,
A=procmail -Y -a $h -d $u
```

Очевидно, удобочитаемость не является сильной стороной sendmail.

Уже давно фирма Microsoft использует язык данных, который может описывать меню, реквизиты окон, диалоговые окна и другие ресурсы Windows. На рис. 2.2 показан фрагмент типичного файла ресурсов. Он читается намного легче, чем пример с программой sendmail, но используется точно так же – компилируется для генерации структуры данных.

Рис. 2.2. Файл .rc для Windows

```
MAIN_MENU MENU
{
```

```
        POPUP " &File"
        {
         MENUITEM "&New", CM_FILENEW
         MENUITEM "&Open…", CM_FILEOPEN
         MENUITEM "&Save", CM_FILESAVE
        }
       }
       MY_DIALOG_BOX DIALOG 6,15,292,287
       STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION |
WS_SYSMENU
       CAPTION "My Dialog Box"
       FONT 8, "MS Sans Serif"
       {
       DEFPUSHBUTTON «OK», ID_OK, 232,16, 50,14
       PUSHBUTTON «Help», ID.HELP, 232, 52, 50,14
       CONTROL "Edit Text Control", ID EDIT1, "EDIT", WS_BODER | WS_TABSTOP,
16,16, 80, 56
       CHECKBOX «Checkbox», ID CHECKBOX 1,153, 65,42,38, BS_AUTOCHECKBOX |
WS_ABSTOP
       }
```

Процедурные языки идут дальше. В этом случае язык является исполняемым и поэтому может содержать инструкции, конструкции управления и т. п. (подобные сценарию на с. 50).

Вы также можете использовать собственные процедурные языки, чтобы облегчить сопровождение программы. Например, вас просят интегрировать информацию из унаследованного приложения в новую разработку графического интерфейса. Обычно это осуществляется при помощи "экранного кармана"; ваше приложение связывается с основным (mainframe) приложением так, как если бы это обычный пользователь-человек, генерируя нажатия клавиш и «считывая» принимаемые отклики. Вы можете создать сценарий взаимодействия при помощи мини-языка [14].

```
locate prompt "SSN:"
type "%s" social_security_number
type enter
waitfor keyboardunlock
if text_at(10,14) is "INVALID SSN" return bad_ssn
if text_at(10,14) is "DUPLICATE SSN" return dup_ssn
# etc…
```

Когда приложение определяет, что пора вводить номер SSN, то по этому сценарию оно вызывает интерпретатор, который затем управляет транзакцией. Если интерпретатор встроен в приложение, то они даже могут совместно использовать данные (например, при помощи механизма обратного вызова).

В этом случае вы программируете в предметной области программиста сопровождения. Когда изменяется основное приложение и поля смещаются, программист может просто обновить высокоуровневое описание, вместо того чтобы копаться в подробностях программы на языке C.

Автономные и встроенные языки

Чтобы приносить пользу, мини-язык не должен использоваться приложением напрямую. Можно многократно использовать язык спецификации для создания искусственных объектов (включая метаданные), которые компилируются, считываются или используются самой программой иным образом (см. "Метапрограммирование").

Например, в разделе "Обработка текста" описывается система, в которой мы использовали Perl, чтобы генерировать большое количество выводов из первоначальной спецификации схемы. Мы изобрели общий язык, чтобы представить схему базы данных, и затем сгенерировали все его формы, которые нам необходимы, – SQL, C, интернет-страницы, XML и др. Приложение не использовало спецификацию напрямую, но оно полагалось на выходные данные, полученные из нее.

Обычной практикой является встраивание процедурных языков высокого уровня непосредственно в ваше приложение, так, чтобы они исполнялись, когда исполняется ваша программа. Очевидно, что это мощное средство; можно изменять поведение приложения, варьируя сценарии, которые оно считывает, причем все это

Несложная разработка или несложное сопровождение?

Мы рассмотрели несколько различных грамматик – от простых строчно-ориентированных форматов до более сложных, которые выглядят как реальные языки. Если для реализации требуются дополнительные усилия, тогда зачем выбирать более сложную грамматику?

Компромиссом являются расширяемость и сопровождение. В то время как программа грамматического разбора «реального» языка может быть более сложной в написании, для пользователя она будет намного понятнее, и ее будет легче расширить за счет добавления новых средств и функциональных возможностей. Слишком простые языки могут быть легкими для грамматического разбора, но они могут быть зашифрованными – подобно примеру с программой sendmail (см. "Языки управления данными и процедурные языки").

Учитывая, что срок службы большинства прикладных программ превышает ожидаемый, вам лучше примириться с суровой действительностью и принять заранее более сложный и удобочитаемый язык. Усилия, затраченные вначале, многократно окупятся за счет снижения затрат на поддержку и сопровождение.

Другие разделы, относящиеся к данной теме:

• Метапрограммирование

Вопросы для обсуждения

• Можно ли выразить некоторые из требований проекта, над которым вы работаете в настоящее время, на языке, отражающем специфику предметной области? Возможно ли написать компилятор или транслятор, который мог бы сгенерировать большую часть требуемой программы?

• Если вы решили принять мини-язык как способ программирования, близкий к предметной области, вы принимаете и то, что для их реализации потребуются некоторые усилия. Как выдумаете, есть ли способы, при которых «скелет», разработанный для одного проекта, может многократно использоваться в других?

Упражнения

5. Требуется реализовать мини-язык управления простым графическим пакетом (возможно, с графикой в относительных командах). Язык состоит из однобуквенных команд. После некоторых команд указывается число. Например, следующий фрагмент изображает на экране прямоугольник. (Ответ см. в Приложении В.)

```
P 2 # select pen 2
D # pen down
W 2 # draw west 2cm
N 1 # then north 1
E 2 # then east 2
S 1 # then back south
U # pen up
```

Составьте программу, которая анализирует этот язык. Она должна быть разработана так, чтобы операция добавления новых команд была несложной.

6. Спроектируйте грамматику BNF (нормальной формы Бэкуса-Наура), чтобы провести грамматический разбор спецификаций времени. Все указанные примеры должны быть успешно проанализированы. (Ответ см. в Приложении В.)

4pm, 7:38pm, 23:42, 3:16, 3:16am

7. Реализуйте программу грамматического разбора для грамматики нормальной формы Бэкуса-Наура в упражнении 6, используя программы yacc, bison или аналогичный генератор грамматического разбора. (Ответ см. в Приложении В.)

8. Реализуйте программу грамматического разбора времени, используя Perl. (Подсказка: регулярные выражения позволяют написать хорошие программы грамматического разбора.) (Ответ см. в Приложении В.)

13
Оценка

Сколько времени потребуется для пересылки "Войны и мира" по модемной линии в 56 байт? Какое место займет на диске миллион имен и адресов? Сколько времени понадобится для прохождения 1000-байтового блока через маршрутизатор? Сколько месяцев потребуется, чтобы завершить ваш проект?

С одной стороны, все эти вопросы бессмысленны – информация, содержащаяся в них, недостаточна для ответа. И тем не менее, на все из них можно ответить, если вы сможете провести оценку. В процессе работы над генерацией оценки, вы придете к большему пониманию мира, в котором обитают ваши программы.

Научившись оценивать и развивая этот навык до уровня, на котором у вас появляется интуитивное ощущение величины того или иного предмета, вы сможете показать явно магическую способность к определению их выполнимости. Если кто-либо говорит: "Мы вышлем вам резервную копию по каналу ISDN в центральный офис", вы сможете интуитивно осознать, имеет ли это смысл. Когда вы составляете программу, вы сможете понять, какие подсистемы нуждаются в оптимизации, а какие нужно оставить в покое.

Подсказка 18: Проводите оценки во избежание сюрпризов

В конце данного раздела мы приведем единственно правильный ответ (в виде бесплатного приложения), который необходимо давать во всех случаях, когда вас просят оценить что-либо.

Насколько точной является "приемлемая точность"?

До некоторой степени все ответы представляют собой оценки. Просто некоторые из них точнее остальных. Так что первым вопросом, который вам придется задать самому себе, когда кто-либо просит вас об оценке, является вопрос о контексте, в котором будет приниматься данный вами ответ. Нужна ли здесь высокая точность, или речь идет о примерной цифре?

• Если ваша бабушка спрашивает, когда вы появитесь, она, вероятно, задается вопросом, готовить вам обед или ужин. С другой стороны, водолаз, оказавшийся в подводной ловушке и испытывающий недостаток воздуха, интересуется ответом с точностью до секунды.

• Каково значение числа «пи»? Если вас интересует, какое количество бордюрного камня понадобится для оформления цветочной клумбы, то цифра 3 вероятно будет приемлемой [15]. На школьном уровне хорошим приближением является 22/7. Ну а если вы работаете в NASA, то двенадцати цифр после запятой будет вполне достаточно.

Одной из интересных особенностей оценки является тот факт, что интерпретация ее результата зависит от используемых вами единиц измерения. Если выговорите, что для некоего действия потребуется 130 рабочих дней, то люди будут ожидать наступления этого события в достаточно узком интервале. Но если вы скажете "около шести месяцев", они будут знать, что этого события следует ожидать через 5–7 месяцев. Обе цифры обозначают одну и ту же продолжительность, но "130 дней", вероятно, подразумевает большую точность, чем вы полагаете. Мы рекомендуем следующую градацию оценок времени:

Продолжительность == Оценка (порядок)

1-15 дней == дни
3-8 недель == недели
8-30 недель == месяцы
30 и более недель == перед тем, как оценить, стоит хорошенько подумать

Так, если после всей необходимой работы, вы придете к решению, что проект займет 125 рабочих дней (25 недель), он может быть оценен как "примерно за шесть месяцев".

Те же принципы применимы к оценкам любых количеств: выберите единицы, в которых будет дан ответ, чтобы отразить точность, которую вы намерены передать.

Из чего исходят оценки?

Все оценки основаны на моделях проблемы. Но перед тем как углубиться в методики построения моделей, необходимо упомянуть о главной хитрости, которая всегда дает хорошие результаты: спросите того, кто уже делал это. Перед тем как вплотную заняться построением модели, оглянитесь вокруг в поисках тех, кто ранее находился в подобной ситуации. Посмотрите, как они решали свою задачу. Маловероятно, что вы обнаружите точное совпадение, но будете удивлены, сколь часто вы успешно обращались к опыту других.

Понимание сути заданного вопроса

Первой частью любого упражнения в составлении оценки является понимание сути заданного вопроса. Как и в случае с вопросами точности, обсуждаемыми выше, вам необходимо осознать масштаб предметной области. Зачастую он неявно выражен в самом вопросе, но осознание масштаба, перед тем как начать строить предположения, должно войти у вас в привычку. Зачастую выбранная вами предметная область частично формирует ответ, который вы даете: "Если предположить, что по дороге не будет аварий и машина заправлена, я буду там через 20 минут".

Построение модели системы

Эта часть процесса оценки – самая интересная. Исходя из вашего понимания заданного вопроса, постройте в уме скелет действующей модели. Если вы оцениваете время отклика, то в вашей модели может иметься узел обслуживания и некий входной поток. При работе над проектом моделью могут послужить стадии, которые ваша организация использует в разработке, наряду с весьма грубым представлением того, как система может быть реализована.

Построение модели может быть творческим процессом, полезным в долгосрочной перспективе. Зачастую построение модели приводит к открытию схем и процессов, лежащих в основе чего-либо и не видимых невооруженным глазом. У вас даже может возникнуть желание повторно исследовать исходный вопрос: "Вы попросили дать оценку X. Однако, похоже, что У, являющийся вариантом X, может быть выполнен примерно в два раза быстрее, и при этом вы теряете лишь одну характеристику".

Построение модели вносит погрешности в процесс оценки. Это и неизбежно, и полезно. Вы жертвуете простотой модели ради точности. Удвоение усилий, прилагаемых к модели, может увеличить точность лишь незначительно. Ваш опыт подскажет вам, когда закончить процесс совершенствования.

Декомпозиция модели

Как только у вас появляется модель, вы можете провести ее декомпозицию на отдельные компоненты. Вам понадобится отыскать математические правила, описывающие взаимодействие этих компонентов. Иногда вклад компонента в конечный результат выражается одной величиной. Некоторые компоненты могут объединять несколько факторов, тогда как

другие могут быть более сложными (подобно тем, которые имитируют поток, приходящий к узлу).

Вы обнаружите, что обычно каждый компонент будет обладать параметрами, которые определяют его влияние на модель в целом. На этой стадии достаточно просто обозначить каждый параметр.

Присвоение значения каждому параметру

Как только в вашем распоряжении появились параметры, вы можете пойти напролом и присвоить некое значение каждому из них. На этой стадии вы ожидаете внесения некоторой ошибки. Хитрость состоит в том, чтобы понять, какие параметры оказывают максимальное воздействие на результат, и сосредоточиться на их точном получении. Обычно параметры, чьи значения добавляются к результату, являются менее значительными, чем те, что осуществляют умножение или деление. Удвоение скорости канала связи может увеличить вдвое объем данных, получаемых в течение часа, тогда как добавление транзитной задержки, равной 5 мс, не даст заметного эффекта.

У вас должен иметься обоснованный способ вычисления этих критических параметров. В примере с формированием очереди вы захотели измерить реальную интенсивность входного потока транзакций в существующей системе или найти для измерения подобную систему. Аналогично, вы могли определить время, необходимое для обслуживания запроса, или провести оценку, используя методики, описанные в этом разделе. На самом деле, вам часто придется основывать свою оценку на других вспомогательных оценках. Именно в этом месте и возникают самые большие ошибки.

Вычисление ответов

Только в самом простом случае ваша оценка будет иметь один-единственный ответ. Вы счастливый человек, если можете сказать: "Я могу пройти по городу пять кварталов за 15 минут". Но поскольку системы все усложняются, вам захочется подстраховать ваши ответы. Проведите многократные вычисления, изменяя значения критических параметров, пока не выясните, какие из них действительно управляют моделью. Серьезную помощь в этом может оказать электронная таблица. Затем сформулируйте ваш ответ с точки зрения этих параметров. "Время отклика составляет (грубо) три четверти секунды, если система имеет шину SCSI и объем памяти 64 Мбайт; и одну секунду при объеме памяти 48 Мбайт". (Заметьте, что "три четверти секунды" дает иное ощущение точности, нежели 750 мс.)

Уже на стадии вычислений появляются ответы, которые могут показаться странными. Не спешите игнорировать их. Если ваша арифметика правильна, то, вероятно, ваше понимание проблемы или модель неверны. Это ценная информация.

Отслеживание уровня мастерства

Мы полагаем, что было бы здорово вести учет ваших оценок так, чтобы вы могли оценить, насколько точным был ваш прогноз. Если общая оценка включала в себя вспомогательные, учитывайте и их. Часто будет оказываться, что ваши оценки удачны – на самом деле, спустя некоторое время вы придете к этому.

Если оценка оказывается неверной, не стоит пожимать плечами и уходить. Стоит выяснить, почему она отличалась от предполагаемой. Возможно, выбраны параметры, не соответствовавшие реальной проблеме. Возможно, сама модель была неверной. Какова бы ни была причина, необходимо не спеша прояснить, что же случилось. Если сделать это, то следующая оценка будет лучше.

Оценка графиков выполнения проекта

Обычные правила оценки могут нарушаться перед лицом сложностей и капризов разработки серьезной прикладной программы. Мы считаем, что зачастую единственным способом определения графика выполнения проекта является практический опыт, полученный при работе над этим проектом. Это не обязательно парадокс, если вы практикуете разработку с помощью приращений, повторяя следующие шаги.

• Проверить требования
• Проанализировать риск
• Осуществить проектирование, реализацию, интеграцию
• Проверить правильность при работе с пользователями

Первоначально у вас может быть лишь приблизительная оценка того, сколько итераций понадобится или какова будет их продолжительность. Некоторые методы требуют, чтобы вы зафиксировали это как часть первоначального плана, но для всех методов, за исключением наиболее тривиальных, это будет ошибкой. Если вы не создаете приложение, аналогичное предыдущему, с той же командой и по той же технологии, вам придется делать предположения.

Итак, вы завершаете составление текста программы и проверку исходной функциональной возможности и отмечаете это как конечную точку первого приращения. Основываясь на этом опыте, вы можете уточнить ваше начальное предположение о числе итераций и о том, что может быть включено в каждую из них. С каждым разом уточнение становится все совершеннее, и вместе с этим растет уверенность в правильности графика.

Подсказка 19: Уточняйте график проекта на основе текста программы

Это может не понравиться руководству, которому обычно нужна единственная надежная цифра еще до начала проекта. Вам придется помочь им осознать, что команда, ее производительность и среда будут определять график выполнения. Формализуя эту процедуру и уточняя график (что является частью итерационного процесса), вы сможете дать руководству самые точные оценки графика выполнения, какие только сможете.

Что сказать, если вас просят оценить что-либо

Говорите "Я вернусь к вам с этим позже".

Вы почти всегда можете добиться лучших результатов, если не будете торопиться и потратите некоторое время, чтобы пройтись по всем стадиям, описанным в данном разделе. К оценкам, сделанным на ходу (например, у офисной кофеварки) придется возвращаться вновь и вновь (как, впрочем, и к кофе), теряя при этом покой.

Другие разделы, относящиеся к данной теме:

• Скорость алгоритма

Вопросы для обсуждения

• Заведите журнал регистрации сделанных вами оценок. Для каждой оценки укажите,

насколько точной она оказалась. Если отклонение превысило 50 %, постарайтесь выяснить, где была допущена ошибка.

Упражнения

9. Спрашивается: какой из двух каналов обладает более широкой полосой пропускания: линия связи со скоростью 1 Мбайт/сек или человек, двигающийся от компьютера к компьютеру со стриммерной кассетой объемом 4 Гбайт в кармане? Какие ограничения накладываются на ответ, чтобы гарантировать его корректность в определенной области? (Например, можно указать, что временем доступа к ленте можно пренебречь.) (Ответ см. в Приложении В.)

10. Так какой же из двух каналов обладает более широкой полосой пропускания? (Ответ см. в Приложении В.)

Глава 3
Походный набор инструментов

Каждый ремесленник отправляется на поиски заработка, имея при себе походный набор инструментов. Столяру могут пригодиться линейки, шаблоны, пара ножовок, несколько рубанков, тонкие стамески, сверла и зажимы, киянки и струбцины. Эти инструменты будет он будет тщательно выбирать и настраивать, каждому из них будет уготована определенная работа, и, что наверное самое важное, каждый из них, оказавшись в умелых руках столяра, найдет свое место под солнцем.

После этого придет черед обучению и притирке. Каждому инструменту будут присущи свои особенности (и хитрости), и каждый из них потребует, чтобы с ним обращались по-своему. При работе столяр держит каждый инструмент особым образом и затачивает его под особым углом. Пройдет время, и от работы инструмент износится до того, что рукоятка превратится в слепок руки столяра, а режущая поверхность сравнится с углом, под которым столяр держит инструмент относительно рабочей плоскости. В этот момент инструменты станут проводниками идей от головы столяра к конечному продукту – они станут продолжением рук мастера. Со временем в арсенале столяра прибавятся новые орудия – резальные машины, лазерные станки для резки под углом, направляющие шаблоны "ласточкин хвост" – все это чудеса технологического прогресса. Но можно поспорить, что по-настоящему столяр счастлив только тогда, когда держит в руках инструмент из старого походного набора и слышит, как рубанок поет свою песню, выстругивая деревянную заготовку.

Инструменты – средство усиления вашего таланта. Чем они лучше и чем лучше вы ими владеете, тем больше вы сможете сделать. Начните с походного универсального набора инструментов. По мере того как вы приобретаете опыт и сталкиваетесь с специальными требованиями, ваш набор пополняется. Стоит уподобиться ремесленнику и пополнять набор регулярно. Старайтесь не прекращать поисков лучшего способа сделать что-либо. Оказавшись в ситуации, когда вы обнаруживаете, что ваших инструментов недостаточно, поищите иное, возможно, более мощное средство для осуществления задуманного. Ваши приобретения должны исходить из существующей необходимости.

Многие начинающие программисты делают ошибку, принимая на вооружение одно-единственное мощное инструментальное средство, в частности, конкретную интегрированную среду разработчика (ИСР), и никогда не выходят за пределы удобного для них интерфейса. Это ошибка. Необходимо осваиваться и вне пределов, установленных ИСР. Но это можно сделать лишь при условии, что инструменты из походного набора должным образом заточены и готовы к работе.

Данная глава посвящена тому, что вкладывается в походный набор инструментов. Как и в любой хорошей дискуссии об инструментах, начнем (в разделе "Преимущества простого текста") с рассмотрения сырья – материала, которому будет придана форма. Затем мы перейдем к верстаку – в нашем случае его роль играет компьютер. Как использовать компьютер для извлечения максимальной пользы из инструментальных средств, находящихся под рукой? Этот аспект обсуждается в разделе "Игры с оболочками". Теперь, когда у нас есть материал и верстак, на котором можно работать, обратимся к инструменту, который, вы наверняка будете использовать чаще всего, – вашему текстовому редактору. В разделе "Мощь редактирования" предлагаются различные способы, как сделать работу с ним более эффективной.

Даже для таких простых вещей, как личная адресная книжка, необходимо использовать

"Систему управления исходным текстом" как гарантию того, что даже самая малая часть вашей драгоценной работы не канет в небытие! И поскольку открыватель законов Мерфи все же был оптимистом, вы не можете считать себя великим программистом, пока не приобретете серьезных навыков в отладке (см. раздел "Отладка").

Чтобы как-то объединить большую часть элементов магии, необходимо некое связующее вещество (наподобие столярного клея). Некоторые средства, подобные awk, Perl и Python, рассмотрены в разделе "Обработка текста".

Подобно тому, как при изготовлении сложных конструкций столяры иногда пользуются шаблонами, программисты могут написать программу, которая, в свою очередь, сама генерирует текст программы. Этот вопрос обсуждается в разделе "Генераторы исходного текста".

Уделив некоторое время изучению этих инструментальных средств, в один прекрасный день вы удивитесь, как ваши пальцы бегают по клавиатуре, обрабатывая текст без дополнительной нагрузки на мозг. Инструменты стали продолжением ваших рук.

14
Преимущества простого текста

Основной материал, с которым работают программисты-прагматики, – не дерево и не металл, а человеческое знание. Оно является форматом при сборе требований, а затем выражается в конструкциях, реализациях, тестах и документации.

И мы уверены, что лучшим форматом для постоянного хранения знания является простой текст, позволяющий обрабатывать знание как вручную, так и с помощью программных средств, используя практически все инструменты, имеющиеся у нас под рукой.

Что такое простой текст?

Простой текст состоит из печатных символов и представлен в некой форме, которая непосредственно может быть воспринята и понята людьми. Например, данный фрагмент не несет в себе смысла, хотя и состоит из печатных символов.

```
Field19=467abe
```

Читатель и понятия не имеет, каков смысл значения 467aЬe. Лучше сделать его понятным:

```
DrawingType=UMLActivityDrawing
```

Простой текст вовсе не означает, что в нем отсутствует структура; яркими примерами простого текста с четко определенной структурой являются форматы XML, SGML и HTML. С простым текстом можно проделывать все те же операции, что и с двоичным форматом, включая управление версиями.

Простой текст имеет тенденцию находиться на более высоком уровне, чем простая двоичная кодировка, обычно возникающая непосредственно из реализации. Предположим, вам нужно хранить свойство под названием usesjnenus, которое может принимать значение TRUE или FALSE. Используя простой текст, вы можете записать это следующим образом:

```
myprop.uses_menus=FALSE
```

А теперь сравните это с 0010010101110101.

Проблема большинства двоичных форматов состоит в том, что контекст, необходимый для понимания данных, отделен от самих данных. Вы искусственно отделяете данные от их смыслового значения. Вдобавок, данные могут быть зашифрованы; они абсолютно бессмысленны при отсутствии прикладной логики для их анализа. А с помощью простого текста вы можете создать самодокументированный поток данных, не зависящий от прикладной программы, которая его породила.

Подсказка 20: Сохраняйте знания в формате простого текста

Недостатки

Простой текст обладает двумя основными недостатками: (1) при хранении он может занимать больше места, чем сжатый двоичный формат, и (2) с точки зрения вычислений интерпретация и обработка файла с простым текстом может проводиться медленнее.

В зависимости от приложения неприемлемыми могут оказаться одна или обе вышеописанные ситуации – например, при хранении данных спутниковой телеметрии или в случае внутреннего формата реляционной базы данных.

Но и в этих ситуациях допустимо сохранять метаданные, описывающие исходные данные, в формате простого текста (см. раздел "Метапрограммирование").

Некоторые разработчики боятся помещать метаданные в формате простого тек ста, потому что таким образом они раскрывают его содержимое пользователям системы. Эти опасения не имеют достаточных оснований. Двоичные данные могут быть более расплывчатыми, чем простой текст, но от этого не становятся более защищенными. Если вы не хотите, чтобы пользователи видели пароли, зашифруйте их. Если вы не хотите, чтобы они изменяли параметры конфигурации, примените технологию защищенного хеширования [16] ко всем значениям параметров, и используйте результат в контрольной сумме файла.

Преимущества простого текста

Поскольку «больше» и «медленнее» – не самые популярные требования, предъявляемые пользователями, то зачем вообще нужен простой текст? Каковы его преимущества?

- Гарантия того, что данные не устареют
- Более короткий путь к цели
- Более простое тестирование

Гарантия того, что данные не устареют

Форматы данных, которые может воспринять человек, и самодокументированные данные переживут все другие форматы данных и приложения, их породившие. И точка.

На протяжении всего срока жизни данных вы сможете пользоваться ими и в перспективе – еще долго после того, как прикладная программа, их породившая, прекратит свое существование.

Вы сможете провести синтаксический анализ такого файла, даже не зная полностью его формата; в большинстве же случаев с двоичными файлами успешный анализ возможен лишь при знании всех особенностей формата.

Рассмотрим файл данных из некой унаследованной [17] системы. Вы обладаете скудной информацией о прикладной программе, которая создала этот файл; эта информация сводится к тому, что она поддерживала список номеров SSN (Social Security Number – номер социального страхования) клиентов, которые вам необходимо найти и извлечь из архива. Среди данных вы видите:

```
«FIELD»123-45-6789«/FIELD10»
:
«FIELD»567-89-0123«/FIELD10»
:
«FIELD10»901-23-4567«/FIELD10»
```

Опознав формат номера SSN, можно быстро написать небольшую программу для извлечения этих данных – даже при отсутствии у вас иной информации об этом файле.

Но представим, что вместо этого файл отформатирован следующим образом:

AC27123456789B11P
:
XY43567890123QTYL
:
6T2190123456788AM

Не так-то легко распознать значение чисел, представленное в таком виде. В этом и состоит разница между воспринимаемым человеком и понятным человеку. Но и от обозначения FIELD10 толку будет немного. А нечто вроде:

«SSNO»123-45-6789«/SSNO»

делает сие упражнение задачкой для детского сада и гарантирует, что данные переживут любой проект, их породивший.

Более короткий путь к цели

Практически любой инструмент в компьютерной вселенной – от систем управления исходными текстами до компиляторных сред, редакторов и отдельно стоящих фильтров – может работать с простым текстом.

Философия ОС Unix

Операционная система Unix известна тем, что она проектировалась на основе философии небольших, отточенных инструментальных средств, каждое из которых предназначено для качественного выполнения только одной операции. Эта философия реализуется с помощью обычного фундаментального формата – строчно-ориентированного файла с простым текстом. Базы данных, используемые в системном администрировании (учетные записи и пароли, конфигурация сети и т. д.), хранятся в виде файлов с простым текстом. (Некоторые системы, подобные Solaris, также поддерживают двоичную форму конкретных баз данных для оптимизации производительности. Версия с простым текстом сохраняется в качестве интерфейса к двоичной версии.)

Если в системе происходит аварийный отказ, то при ее восстановлении может оказаться, что вам придется работать в среде с минимальным интерфейсом (например, вы не будете иметь доступа к графическим драйверам). Подобные ситуации дают возможность оценить простоту выбранного представления текста.

Предположим, что вы развертываете крупномасштабное приложение со сложным конфигурационным файлом, характерным для конкретного местоположения (на ум сразу приходит sendmail). Если этот файл представляет собой простой текст, то его можно подчинить системе управления исходными текстами (см. "Управление исходным текстом"), и вы

автоматически сохраняете хронологию всех изменений. Инструментальные средства сравнения файлов, такие как diff и fc, позволяют сразу увидеть, какие изменения были внесены, тогда как sum позволяет генерировать контрольную сумму для отслеживания файла на предмет случайных (или злонамеренных) модификаций.

Более простое тестирование

Если вы используете простой текст при создании синтетических данных для запуска системных тестов, то добавление, обновление или модификация тестовых данных (без привлечения каких-либо специальных инструментальных средств) не представляет особого труда. Аналогично, результат регрессионного тестирования в виде простого текста может быть проанализирован тривиальным образом (например, с помощью программы diff) или более тщательно с помощью языков Perl, Python и при помощи некоторых других средств написания сценариев (скриптов).

Подводим итог

Вездесущий текстовый файл никуда не денется и в будущем, когда интеллектуальные XML-базированные агенты, путешествующие по диким и опасным дебрям Интернета в автономном режиме, будут согласовывать обмен данными между собой. Действительно, в гетерогенных операционных средах преимущества простого текста могут перевесить все его недостатки. Необходимы гарантии того, что все стороны могут обмениваться информацией по общему стандарту. Таким стандартом и является простой текст.

Другие разделы, относящиеся к данной теме:

- Управление исходным текстом
- Генераторы исходного текста
- Метапрограммирование
- Доски объявлений
- Вездесущая автоматизация
- Все эти сочинения

Вопросы для обсуждения

• Требуется спроектировать базу данных – небольшую адресную книгу (фамилия, номер телефона и т. д.), используя простое двоичное представление на языке по вашему выбору. Перед тем как продолжить чтение данного проблемного вопроса, проделайте следующее:

1. Преобразуйте этот формат в формат простого текста, используя XML.

2. Для каждой из версий добавьте новое поле переменной длины под названием directions, в котором вы могли бы вводить указания, как подъехать к дому каждого адресата.

Какие вопросы, связанные с управлением версиями и расширяемостью, могут возникнуть? Какую форму легче модифицировать? Как обстоит дело с преобразованием уже существующих данных?

15
Игры с оболочками

Каждому столяру нужен хороший, солидный, надежный верстак – место, расположенное на удобной для столяра высоте, на котором он в ходе своей работы мог бы разместить предметы труда. Верстак становится центром мастерской, столяр возвращается к нему снова и снова, придавая форму материалу.

Для программиста, обрабатывающего файлы или текст, подобным верстаком является командная оболочка. Находясь в командной строке, вы можете задействовать весь свой арсенал инструментов, комбинируя их такими способами, о которых их разработчики и не мечтали. Из оболочки вы можете запускать приложения, отладчики, браузеры, редакторы и утилиты. Вы можете осуществлять поиск файлов, опрашивать состояние системы и производить фильтрацию выходных данных. Для часто используемых процедур вы можете создавать сложные макрокоманды, используя встроенный язык.

Для программистов, выросших на графических интерфейсах и ИСР, это может показаться экстремизмом. В конце концов, разве нельзя проделать все операции с равным успехом, указывая на объект и щелкая кнопкой мыши?

Ответ прост: «Нет». Графические интерфейсы сами по себе прекрасны, и с их помощью многие простые операции выполняются быстрее и с большим удобством. Перемещение файлов, чтение сообщений электронной почты с кодировкой MIME и набор текстов писем – это все то, что вы хотели бы осуществлять в графической среде. Но если выделаете всю работу, используя графический интерфейс, то используете далеко не все возможности, предоставляемые операционной системой. И вам не удастся автоматизировать обычные задачи или использовать доступные инструментальные средства в полную силу. И вы не сможете комбинировать свои средства для создания специализированных макроинструментов. Преимуществом графического интерфейса пользователя является принцип WYSIWYG – что видишь, то и получаешь. Недостатком графического интерфейса можно назвать принцип WYSIAYG – получаешь только то, что видишь.

Графические среды обычно ограничены возможностями, заложенными в них разработчиками. Если вам необходимо выйти за пределы модели, созданной разработчиком, то обычно фортуна отворачивается от вас, однако чаще всего вам все-таки приходится выходить за пределы модели. Прагматики не просто либо «рубят» текст, либо разрабатывают объектные модели, либо пишут документацию или автоматизируют процесс сборки – они делают все вышеперечисленное. Сфера применения любого конкретного инструмента обычно ограничена задачами, решения которых от него ожидают. Предположим, возникла необходимость в интеграции препроцессора исходного текста с ИСР (при реализации концепции "проектирования по контракту", многопроцессных директив, и т. п.). Если разработчик ИСР явно не предусмотрел наличия в ней специальных средств, то вы не справитесь с решением задачи интеграции.

Если вы уже освоились с работой в режиме командной строки, то можете спокойно пропустить данный раздел. В противном случае вам необходимо заручиться дружеским расположением со стороны командной оболочки.

Исповедуя прагматизм, вы постоянно будете испытывать потребность в осуществлении операций ad hoc (лат. для конкретного случая. – Прим. пер.) – это и есть те самые случаи, когда графический интерфейс может оказаться неприменимым. Командная строка может стать лучшим решением, если необходимо быстро скомбинировать несколько команд при выполнении

запроса или иного задания. Ниже приводится несколько примеров.

Найти все файлы типа *.c, модифицированные позже, чем ваш Makefile.

Командная строка:

find. -name *.c' – newer Makefile – print

Графический интерфейс:

Откройте Проводник Windows, перейдите в нужный каталог, щелкните по Makefile и отметьте для себя время модификации данного файла. Затем войдите в меню Tools, выберите пункт Find, и введите *.c в строку, указывающую параметры имени файла. Затем перейдите в поле даты, и введите дату, которую вы вначале отметили для Makefile. Затем нажмите ОК.

Создать архив типа zip/tar моего исходного текста.

Командная строка:

zip archive.zip *.h *.c или

tar cvf archive.tar \h *.c

Графический интерфейс:

Запустите утилиту архивирования (например, условно-бесилатную программу WinZip [URL 41], выберите пункт Create New Archive, введите его имя, выберите исходный каталог в диалоге Add, задайте фильтр "*.c", щелкните по пункту «Add», задайте фильтр "*.h", щелкните по пункту «Add», затем закройте архив.

Какие файлы Java не были изменены за последнюю неделю?

Командная строка:

find . Name *.java' – mtime + 7 – print

Графический интерфейс:

Щелкните и переместитесь к пункту "Find files", щелкните по полю «Named» и введите в него "*.java", выберите пункт "Date Modified". Затем выберите пункт «Between». Затем щелкните по начальной дате и введите начальную дату начала проекта. Щелкните по конечной дате и введите дату, которая была неделю назад (убедитесь, что календарь находится под рукой). Затем щелкните по пункту "Find Now".

Какие из данных файлов используют библиотеки awt?

Командная строка:

find . -name *.java' – mtime +7 – print | xargs grep 'java.awt'

Графический интерфейс:

Загрузите каждый файл в списке из предыдущего примера в редактор и проведите поиск строки java.awt. Напишите имя каждого файла, содержащего совпадение.

Ясно, что этот список может быть продолжен. Строчные команды могут быть непонятными и компактными, но они обладают мощностью и краткостью. И поскольку они могут сводиться в файлы сценариев (или командные файлы в системе Windows), то вы можете создавать последовательности команд для автоматизации часто выполняемых процедур.

Подсказка 21: Используйте сильные стороны командных оболочек

Освойте работу с оболочкой, и вы обнаружите, как выросла ваша производительность.

Нужно создать перечень всех уникальных имен пакетов, которые явно импортируются вашей программой на языке Java? Приведенная ниже программа сохраняет этот перечень в файле под названием "list".

```
grep "import' *.java |
  sed -e's/.'import //' – e's/;.$//' |
  sort -u >list
```

Если вам еще не приходилось часами изучать возможности командной оболочки систем, с которыми вы работаете, то это занятие может показаться устрашающим. Тем не менее, приложите некоторое усилие для ознакомления с оболочкой, и вскоре все встанет на свое место. Поиграйте с вашей командной оболочкой, и вы удивитесь, насколько продуктивнее станет ваша работа.

Утилиты оболочек и системы Windows

Хотя командные оболочки, поставляемые с системами Windows постепенно улучшаются, утилиты командной строки Windows все еще уступают их двойникам в Unix. Однако все не так плохо.

Фирма Cygnus Solutions разработала пакет под названием Cygwin [URL 31]. Помимо обеспечения слоя совместимости Unix для Windows, Cygwin поставляется вместе с коллекцией более чем 120 утилит Unix, включая такие бестселлеры, как Is, grep и find. Утилиты и библиотеки могут загружаться и использоваться бесплатно, но обязательно прочтите их лицензию [18]. Программа Cygwin распространяется вместе с оболочкой Bash.

Использование инструментальных средств Unix при работе в среде Windows

Нам нравится, что высококачественные инструментальные средства Unix работают в среде Windows, и мы пользуемся ими ежедневно. Однако надо иметь в виду, что существуют проблемы интеграции. Эти утилиты (в отличие от их двойников, работающих в MS-DOS) чувствительны к регистру в именах файлов, так что команда ls a*.bat не сможет найти файл AUTOEXEC.BAT. Вы можете также столкнуться с проблемами, вызванными файлами, чьи имена содержат пробелы, и с различиями в разделителях пути. Наконец, есть проблемы, связанные с запуском из-под оболочек Unix программ MS-DOS, в которых ожидается наличие аргументов в стиле MS-DOS. Например, утилиты Java, написанные фирмой JavaSoft, используют двоеточие (как и в их разделителе CLASSPATH при работе в среде Unix), а при работе в MS-DOS используют точку с запятой. В результате сценарий Bash или ksh, запускаемый в окне Unix, будет работать также и в Windows, но командная строка, передаваемая ею Java, будет интерпретироваться некорректно.

В качестве альтернативы Дэвид Корн (автор известной оболочки Korn) создал пакет под названием UWIN. Он предназначен для тех же целей, что и продукт Cygwin – это среда разработчика Unix, работающая в Windows. Пакет UWIN распространяется с оболочкой Korn. Коммерческие версии поставляются фирмой Global Technologies, Ltd. [URL 30]. Кроме того, фирма AT&T допускает бесплатную загрузку пакета для оценки его работы и использования в академических учреждениях. Перед его использованием также необходимо прочесть лицензию.

И наконец, Том Кристиансен (во время написания книги) компонует Perl Power Tools, пытаясь в сжатом виде реализовать все известные утилиты Unix на языке Perl [URL 32].

Другие разделы, относящиеся к данной теме:

• Вездесущая автоматизация

Вопросы для обсуждения:

• Существуют ли операции, которые в данное время вам приходится выполнять вручную, работая в графической среде? Приходилось ли вам когда-либо сочинять для коллег по работе инструкции, состоящие из отдельных пунктов типа: "щелкните по этой кнопке", "выберите этот пункт"? Можно ли автоматизировать данный процесс?

• При переходе к новой операционной среде обратите особое внимание на то, ка – кие оболочки находятся в вашем распоряжении. Посмотрите, можете ли вы перенести в новую среду оболочку, с которой работаете в данный момент.

• Изучите возможные альтернативы оболочке, используемой вами в настоящее время. Если вы сталкиваетесь с проблемой, которую невозможно решить средствами имеющейся у вас оболочки, может быть, альтернативная оболочка проявит себя лучше?

16
Мощь редактирования

Выше уже говорилось об инструментах, которые являются продолжением вашей руки. Это положение применимо к текстовым редакторам в большей степени, нежели к любому другому инструменту. Необходимо, чтобы вы затрачивали минимальные усилия на обработку текста, поскольку последний является основным «сырьем» при программировании. Рассмотрим некоторые общие характеристики и функции, которые помогают использовать ваш текстовый редактор с максимальным КПД.

Один-единственный редактор

Мы полагаем, что лучше овладеть одним-единственным редактором, но в совершенстве, и использовать его для решения всех задач, связанных с редактированием: работа с текстом программ, документацией, записками, системное администрирование и т. д. Не имея под рукой хотя бы одного редактора, можно оказаться в ситуации, аналогичной вавилонскому смешению языков, но уже на современный манер. При написании текстов программ может понадобиться встроенный редактор ИСР (для каждого языка), для создания документации – универсальный офисный редактор, а может быть, и еще один встроенный редактор для отправки сообщений по электронной почте. Различаться могут даже клавиатурные команды, используемые вами для редактирования командных строк в оболочке [19]. Трудно быть экспертом в любой из этих программных сред, если в каждой из них имеется свой набор команд и соглашений при редактировании.

Но экспертом быть необходимо. Мало набирать символы построчно и использовать мышь для вырезания и вставки фрагментов. Работая подобным образом, вы не достигнете того уровня производительности, который возможен при наличии мощного текстового редактора. Десятикратное нажатие клавиши <- или BACKSPACE для перемещения курсора влево к началу строки не столь эффективно, как простая клавиатурная команда, например Ctrl+A, Home или 0.

Подсказка 22: Используйте один текстовый редактор, но по максимуму

Выберите какой-либо редактор, тщательно изучите его и используйте во всех задачах, связанных с редактированием текста. Если вы пользуетесь одним редактором (или набором функциональных клавиш) для всех работ, связанных с редактированием текста, вам не придется останавливаться и обдумывать, как осуществить ту или иную обработку текста: нажатие нужных клавиш становится рефлексом, редактор – продолжением вашей руки: клавиши поют свою песню, перемещаясь по тексту сквозь череду мыслей. Это и есть цель!

Убедитесь, что выбранный вами редактор поддерживается всеми платформами, с которыми вы работаете. Редакторы Emacs, vi, CRISP, Brief и ряд других поддерживаются несколькими платформами, часто в двух версиях – в графической и неграфической (текстовый режим).

Средства редактирования

Помимо тех средств, которые вы считаете особенно полезными и удобными, имеется ряд

основных возможностей, которыми, по нашему мнению, должен обладать любой приличный редактор. Если в вашем редакторе отсутствует любая из этих возможностей, то, вероятно, настало время поразмыслить о переходе к более продвинутому редактору.

• **Настраиваемость.** Все свойства редактора должны настраиваться по вашему пожеланию, включая шрифты, цвета, размеры окон и клавиатурные привязки (команды, исполняемые при нажатии той или иной клавиши). Применение только клавиатурных сочетаний в ходе обычных операций редактирования является более эффективным по сравнению с мышью или командами в меню, поскольку руки не отрываются от клавиатуры.

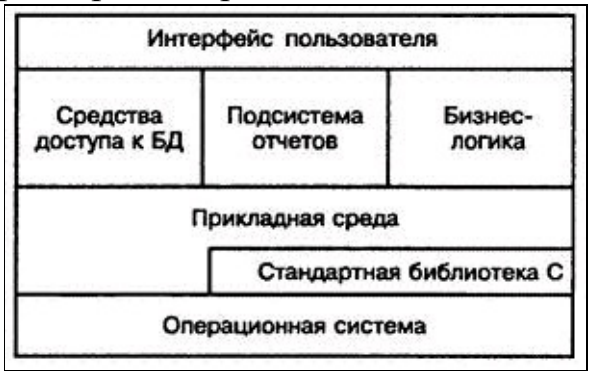
• **Расширяемость.** Редактор не должен устаревать только потому, что появляется новый язык программирования. Он должен обладать способностью интегрироваться в любую компиляторную среду, используемую вами в данный момент. Вы должны «обучить» его нюансам любого нового языка программирования или текстового формата (XML, HTML версии 9, и т. д.).

• **Программируемость.** Вы должны располагать возможностью программирования редактора для осуществления сложных многоступенчатых операций. Это может осуществляться при помощи макросов или встроенного языка программирования сценариев (к примеру, редактор Emacs использует вариант языка Lisp).

В дополнение к этому многие редакторы поддерживают средства, которые свойственны конкретному языку программирования:

• Выделение синтаксических конструкций
• Автоматическое завершение
• Автоматический отступ
• Библиотека исходных стандартных текстов или документов
• Принудительная справка, выдаваемая системой
• Средства, подобные ИСР (компиляция, отладка и т. д.).

Рис. 3.1. Упорядочение строк в редакторе



Такое средство, как выделение синтаксических конструкций, может показаться необязательной фривольностью, но в реальности оно весьма полезно и улучшит вашу производительность. Вы быстро привыкнете к тому, что ключевые слова отображаются разным цветом или шрифтом, а неправильно набранное ключевое слово, которое отображается по-другому, бежит на вас как зверь на ловца задолго до того, как вы запустите компилятор.

Возможность компиляции и обнаружения ошибок непосредственно в среде редактирования – это значительное удобство при работе над большим проектами. Приверженцем подобного стиля взаимодействия является, в частности, редактор Emacs.

Производительность

Удивительно, но многие люди, с которыми нам довелось встречаться, используют для редактирования исходных текстов программ утилиту notepad, входящую в систему Windows. Это сильно напоминает использование чайной ложечки вместо совковой лопаты – примитивный набор текста и использование основных команд ("вырезать-вставить"), активизируемых с помощью мыши, явно недостаточны

Какие же процедуры могут потребовать от вас большего, чем примитивные действия, описанные выше?

Начнем с перемещения курсора. Некоторые клавиатурные команды, перемещающие вас по словам, строкам, блокам или функциям, являются более эффективными по сравнению с повторным нажатием одной и той же клавиши, перемещающей курсор от символа к символу или от строки к строке.

Предположим, что вы пишете программу на языке Java. Вам нравится поддерживать ваши операторы import в алфавитном порядке, но кто-то чужой зарегистрировал несколько файлов, не отвечающих этому стандарту Вы хотели бы пробежать по нескольким файлам и упорядочить некоторую их часть. Это легко сделать при работе с редакторами типа vi и Emacs (см. рис. 3.1). А попробуйте сделать то же самое в редакторе notepad!

Некоторые редакторы могут помочь в усовершенствовании обычных операций. Например, когда вы создаете новый файл на определенном языке программирования, редактор может подобрать для вас нужный шаблон. Он может включать в себя:

• Наименование создаваемого класса или модуля (определенного из имени файла)

• Ваше имя и/или знак авторского права

• «Скелеты» конструкций на данном языке (например, описания конструктора или деструктора)

Другим полезным средством является автоматический отступ. В нужный момент редактор автоматически делает отступы (например, при вводе открывающей скобки), и не нужно проделывать ту же операцию вручную (используя пробел или табуляцию). Преимуществом этого средства является то, что вы можете использовать редактор для обеспечения постоянства стиля отступа в вашем проекте [20].

Куда же направиться?

Советы подобного рода особенно трудно формулировать, поскольку практически все читатели находятся на различных уровнях владения редактором (или редакторами), которым они пользуются в данный момент, и имеют различный опыт работы с ними. Поэтому, чтобы подвести итог и сформулировать некоторые указания относительно того, куда же двигаться дальше, отыщите то, что соответствует вам в левом столбце таблицы, а затем, чтобы узнать наше мнение обо всем этом, взгляните на правый столбец.

Если это похоже на вас:
Тогда задумайтесь о следующем:

Я пользуюсь только основными средствами многих разнообразных редакторов.
Выберите мощный редактор и изучите его хорошенько.

У меня есть любимый редактор, но я не использую все его средства
Изучите эти средства. Сократите число используемых клавиатурных команд.

У меня есть любимый редактор и я использую его везде, где только возможно

Постарайтесь расширить свои познания и используйте его для большего числа задач, по сравнению тем, что вы используете сегодня.

Я думаю, что вы говорите ерунду. Notepad является самым лучшим редактором из когда-либо созданных

Пока вы счастливы и продуктивны, используйте notepad! Но если вы окажетесь подвержены чувству "редакторской зависти", возможно, что вы и пересмотрите свою позицию

Какой же редактор выбрать?

Советуем освоить приличный редактор, но какой же именно? Уклонимся от ответа на этот вопрос: выбор редактора является личным делом каждого (некоторые даже скажут, что выбор редактора связан с вероисповеданием!). В приложении А приведен список популярных редакторов и мест, откуда их можно загрузить.

Вопросы для обсуждения

• Некоторые редакторы используют полномасштабные языки для настройки и создания сценариев. Например, в редакторе Emacs используется язык программирования Lisp. В качестве одного из новых языков, который вы наметили изучить в этом году, изучите язык, используемый вашим редактором. Разработайте набор макросов (или эквивалентных им средств) для всех операций, которые вам приходится осуществлять повторно.

• А знаете ли вы все, на что способен ваш редактор? Попытайтесь подзадорить ваших коллег, которые работают с тем же редактором. Попробуйте выполнить любое задание, связанное с редактированием, используя как можно меньшее число клавиатурных команд.

17
Управление исходным текстом программ

Прогресс не проявляется лишь в изменениях и зависит от цепкости памяти. Те, кто не учится на своих ошибках, обречены повторять их.

Джордж Сантаяна, Жизнь разума

Одним из наиболее важных свойств, которые интересуют нас в интерфейсе пользователя, является кнопка UNDO – единственная кнопка, которая прощает нам наши ошибки. Еще лучше, если графическая среда поддерживает многоуровневый откат и повтор так, что можно вернуться назади восстановить статус-кво, существовавший за несколько минут до этого. Но как быть, если ошибка произошла на прошлой неделе и за прошедшее время компьютер включался и выключался раз десять? Это и является одним из многих преимуществ системы управления исходным текстом программ: она является своего рода гигантской клавишей UNDO – машиной времени, работающей в масштабах проекта, которая способна вернуть вас к безмятежным дням на прошлой неделе, когда программа реально компилировалась и запускалась.

Системы управления исходным текстом (или в более широком смысле системы управления конфигурацией) отслеживают любые изменения, которые вносятся в исходный текст и документацию. Лучшие системы также могут отслеживать версии компилятора и операционной системы. С помощью системы управления исходным текстом, сконфигурированной надлежащим образом, всегда можно вернуться к предыдущей версии программы.

Но система управления исходным текстом (английская аббревиатура SCCS) [21]дает много больше, чем просто отмену ошибочных действий. Хорошая система позволяет отслеживать изменения и дает ответы на характерные вопросы: "Кто внес изменения в данную строку текста? В чем состоит разница между версией, существующей на данный момент, и версией, существовавшей на прошлой неделе? Сколько строк текста программы были изменены в данной версии? Какие файлы изменяются чаще всего?" Подобная информация бесценна при отслеживании ошибок, аудите, оценке производительности и качества.

Система управления также позволяет проводить идентификацию версий программы. После идентификации вы всегда сможете вернуться к нужной версии и восстановить ее, независимо от более поздних изменений.

Системы управления часто используются для работы с ответвлениями в древовидной схеме разработки. Например, после выпуска некоторой программы обычно возникает желание продолжить ее разработку и выпустить новую версию. Но при этом приходится работать над ошибками в текущей версии и передавать заказчикам исправления. Фрагменты с устраненными ошибками должны перейти (если это приемлемо) в последующую версию, но к заказчикам незаконченная программа не должна попасть. Всякий раз, когда вы генерируете версию в целом, при помощи системы управления можно сгенерировать и ответвления в древовидной схеме разработки. Ошибки, имеющиеся в ответвлении, устраняются с одновременным продолжением работ по усовершенствованию ствола. Так как устраняемые ошибки могут иметь отношение и к стволу, то некоторые системы управления позволяют автоматически распространить определенные изменения, сделанные в ответвлении, обратно на ствол древовидной схемы.

Системы управления могут сохранять поддерживаемые ими файлы в централизованной БД проекта – лучшем кандидате на архивирование.

И наконец, некоторые программные продукты позволяют двум и более пользователям

работать одновременно с одним и тем же набором файлов и даже вносить изменения в один и тот же файл одновременно. Затем система управляет слиянием изменений при возвращении этих файлов в централизованную БД проекта. При всей кажущейся рискованности на практике подобные системы полезны в работе с проектами различного масштаба.

Подсказка 23: Всегда используйте управление исходным текстом программы

Всегда. Даже если ваша команда состоит из одного человека и продолжительность проекта составляет одну неделю. Даже если это прототип на выброс. Даже если материал, с которым вы работаете, не является исходным текстом программы. Убедитесь, что все находится под контролем – документация, номера телефонов, записки поставщикам, сборочные файлы, процедуры сборки и выпуска, крохотный сценарий (в оболочке), прожигающий эталонный компакт-диск, словом – все. Обычно мы используем управление исходным текстом в отношении всего того, что мы набираем (включая текст данной книги). И даже если мы не работаем над проектом, каждодневная работа надежно сохраняется в централизованной БД.

Сборки и управление исходным текстом

Если весь проект находится под защитой системы управления исходным текстом, то он обладает огромным скрытым преимуществом: вы можете создавать сборки программы, которые являются автоматическими и воспроизводимыми.

Механизм сборки проекта может автоматически извлекать последнюю версию исходного текста из централизованной БД. Этот механизм может запускаться среди ночи, после того как все сотрудники (будем надеяться на это) уйдут домой. Вы можете автоматически прогонять регрессионные тесты для гарантии того, что исходные тексты, созданные в течение рабочего дня, ничего не нарушили. Автоматизация сборки обеспечивает согласованность – отсутствуют ручные процедуры, и вам не нужно, чтобы разработчики помнили о копировании созданного ими текста в специальную сборочную область.

Сборка является воспроизводимой, так как вы всегда можете заново собрать исходный текст в том виде, в каком он существовал на указанную календарную дату.

Команда, в которой я работаю, не использует систему управления исходным текстом

Как же им не стыдно! Звучит как перспектива провести очередную Реформацию! Однако, пока вы дождетесь, когда они увидят свет во тьме, стоит попробовать внедрить свою, частную систему управления. Воспользуйтесь одним из бесплатных инструментальных средств, указанных в приложении А, и обратите особое внимание на то, чтобы результаты вашей личной работы были надежно сохранены в централизованной БД. Хоть это и может показаться двойной работой, мы с уверенностью можем сказать, что эта процедура сбережет ваши нервы (и сэкономит деньги, отпущенные на проект) в тот момент, когда вам впервые придется ответить на вопросы типа "Что ты натворил с модулем xyz?" и "Кто разрушил сборку?" Подобный подход поможет вам убедить руководство в том, что система управления исходным текстом действительно работает.

Не забывайте, что система управления в равной степени применима и к тому, с чем вы имеете дело помимо основной работы.

Программы управления исходным текстом

В приложении A приведены интернет-ссылки (URL) на типичные системы управления исходным текстом – некоторые из них являются коммерческими продуктами, другие же распространяются бесплатно. Имеются и другие программные продукты – обратите внимание на ссылки на часто задаваемые вопросы (FAQ) по управлению конфигурацией.

Другие разделы, относящиеся к данной теме:

- Ортогональность
- Преимущество простого текста
- Все эти сочинения

Вопросы для обсуждения

• Даже если у вас нет возможности использовать систему управления исходным текстом на работе, установите RCS или CVS на личный компьютер. Воспользуйтесь ей для управления вашими домашними проектами, документами, которые вы составляете, и (возможно) изменениями в конфигурации самой компьютерной системы.

• Обратите внимание на некоторые из проектов с открытыми исходными текстами, архивы которых доступны в сети Интернет (например, Mozilla [URL 51], KDE[URL 54] и Gimp [URL 55]). Каким образом вы получаете обновления исходного текста? Как вы вносите изменения – сам проект регулирует доступ, или же разрешает внесение изменений?

18
Отладка

Смотреть в себя, зреть муки свои, Зная, что сам ты виновник мук, – Вот истинное страданье.

Софокл, Аякс

Английское слово bug (ошибка) используется для описания "объекта, вызывающего ужас" уже начиная с XIV в. Контр-адмирал д-р Грэйс Хоппер (создатель языка COBOL) оказался первым, кто наблюдал компьютерного «жучка», буквально – моли, попавшей в одно из электромеханических реле, из которых состояли первые вычислительные системы. Когда техника попросили объяснить, почему машина ведет себя не так, как надо, он сообщил, что в системе "завелся жучок", и в соответствии со своими должностными обязанностями приклеил его клейкой лентой вместе с крылышками и всем остальным в рабочий журнал.

К сожалению, мы до сих пор встречаемся с «жучками» в системе, хотя и не из рода перепончатокрылых. Но значение этого слова, принятое в XIV в. – привидение – возможно более применимо сейчас, нежели тогда. Изъяны в программном обеспечении проявляют себя по-разному – от превратно истолкованных требований до ошибок в написании исходных текстов. К сожалению, возможности современных компьютерных систем все еще ограничены исполнением только того, что мы им прикажем, а не обязательно того, что мы хотим, чтобы они сделали.

Никто не создает совершенное программное обеспечение, так что примите как данность тот факт, что отладка будет занимать большую часть вашего рабочего дня. Рассмотрим некоторые аспекты, вовлеченные в процесс отладки, и некоторые универсальные стратегии поиска неуловимых ошибок.

Психология процесса отладки

Сама по себе отладка является щепетильным и нервирующим моментом для многих разработчиков. Вместо того, чтобы наброситься на нее, как на головоломку, которая должна быть решена, вы можете встретиться с отрицанием, неубедительными отговорками и просто апатией.

Воспользуйтесь тем фактом, что отладка представляет собой не что иное, как решение задачи, и атакуйте ее именно с этой позиции.

Обнаружив чью-то ошибку, вы можете тратить время и силы на обвинения мерзкого преступника, ее допустившего. В некоторых сферах деятельности это является частью культуры и обладает свойством катарсиса. Однако в технической сфере вы хотите сконцентрироваться на устранении проблемы, а не на выяснении, кто виноват.

Подсказка 24: Занимайтесь устранением проблемы, а не обвинениями

На самом деле, не важно, кто виноват в ошибке – вы или кто-то другой. Это все равно остается вашей проблемой.

Умонастроение отладки

Обманывать самого себя легче всего.

Эдвард Булвер-Литтон, Отвергнутый

Перед тем как начать отладку, важно настроиться. Необходимо отключить многие средства безопасности, которые вы ежедневно используете для защиты собственного «я», сбросить проектный прессинг, под которым вы можете находиться, и успокоиться. Прежде всего помните первое правило отладки:

Подсказка 25: Не паникуйте

Легко впасть в панику, особенно если вы связаны контрольными сроками или работаете с нервным руководителем или заказчиком, стоящим у вас над душой в то время, когда вы пытаетесь найти причину ошибки. Но очень важно сделать шаг назад и подумать над тем, что же на самом деле является первопричиной симптомов, которые, по вашему убеждению, являются ошибкой.

Если ваша первая реакция после обнаружения ошибки или просмотра отчета об ошибках сводится к восклицанию "Это невозможно!", то вы явно ошиблись. Не стоит тратить ни одного нейрона на цепочку умозаключений, начинающуюся с фразы "Но этого не может быть!", потому что совершенно ясно, что может, и это произошло.

Остерегайтесь близорукости во время отладки. Воспротивьтесь желанию устранить лишь те признаки, которые видны невооруженным глазом: скорее всего, действительная причина может находиться в нескольких шагах от того, что вы наблюдаете, и может включать ряд сопутствующих проблем. Всегда пытайтесь обнаружить глубинную причину проблемы, а не ее частное проявление.

С чего начать?

Перед тем как взглянуть на ошибку, убедитесь, что вы работаете над программой, которая прошла стадию компиляции чисто – без предупреждений. Обычно мы устанавливаем уровни предупреждения компиляторов максимально высокими. Нет смысла тратить время в попытках найти проблему, которую не смог найти и компилятор! Необходимо сосредоточиться на более сложных насущных проблемах.

Пытаясь решить любую проблему, нужно собрать все относящиеся к делу данные. К сожалению, отчеты об ошибках не являются точной наукой. Легко впасть в заблуждение из-за совпадений, а вы не можете позволить себе тратить время на исследование причин совпадений. Необходимо быть точным в ваших наблюдениях изначально.

Точность отчетов об ошибках снижается еще больше, когда их просматривает третья сторона, в реальности может оказаться, что вам придется наблюдать за действиями пользователя, который сообщил об ошибке, чтобы добиться достаточного уровня детализации.

Однажды один из авторов книги (Энди Хант) работал над большим графическим приложением. Дело уже шло к выпуску готовой версии, когда тестировщики сообщили о том, что приложение «падало» всякий раз, когда они проводили черту при помощи конкретной

кисти. Программист начал оспаривать это утверждение, говоря о том, что все в порядке: он сам пытался выполнять аналогичную прорисовку, и все работало превосходно. Обмен любезностями продолжался в течение нескольких дней, когда напряженность вдруг резко возросла.

В конце концов все собрались в одной комнате. Тестировщик выбрал нужный инструмент (кисть) и провел черту, из ВЕРХНЕГО ПРАВОГО угла к НИЖНЕМУ ЛЕВОМУ. Приложение «упало»! Программист тихонько охнул, а затем виновато проблеял, что при тестировании он проводил черту только из НИЖНЕГО ЛЕВОГО угла к ВЕРХНЕМУ ПРАВОМУ, и при этом ошибка никак не выявлялась.

В этой истории есть два момента, заслуживающих внимания:

• Может возникнуть необходимость в опросе пользователя, который сообщил о присутствии ошибки, для того чтобы собрать больше данных, чем было дано изначально.

• Искусственные тесты (такие, как одна-единственная черта, проведенная «кистью» снизу вверх) недостаточны для испытания приложения. Необходимо осуществлять тестирование обоих граничных условий и реалистических шаблонов действия конечного пользователя. Это нужно делать систематически (см. "Безжалостное тестирование").

Стратегии отладки

Если вы уверены, что знаете, в чем дело, пора выяснить, как сама программа относится к происходящему.

Воспроизведение ошибок

Нет, наши ошибки на самом деле не размножаются (хотя некоторые из них возможно достаточно стары, чтобы делать это уже на законных основаниях). Мы говорим о другом способе размножения.

Начать устранение ошибки лучше всего с придания ей свойства воспроизводимости. В конце концов, если вы не можете воспроизвести ее, то как узнать, что она вообще устранена?

Но нам нужно нечто большее, чем ошибка, которая воспроизводится с помощью некоторой последовательности операций; нам нужна ошибка, которую можно воспроизвести при помощи одной-единственной команды. Процедура устранения ошибки многократно усложняется, когда вам приходится выполнять 15 операций, чтобы добраться до места, где эта ошибка выявляется. В ряде случаев вы можете интуитивно понять, как можно устранить ошибку, заставив себя абстрагироваться от тех обстоятельств ее проявления.

Другие идеи, касающиеся вышеприведенного, представлены в разделе "Вездесущая автоматизация".

Сделайте ваши данные наглядными

Пристальный взгляд на данные, с которыми работает программа, во многих случаях является лучшим способом увидеть то, что же она делает (или собирается делать). Простейшим примером этого является прямолинейный подход типа "переменная = значение", который может быть реализован в виде печатного текста или в виде полей диалогового окна (списка) графического интерфейса.

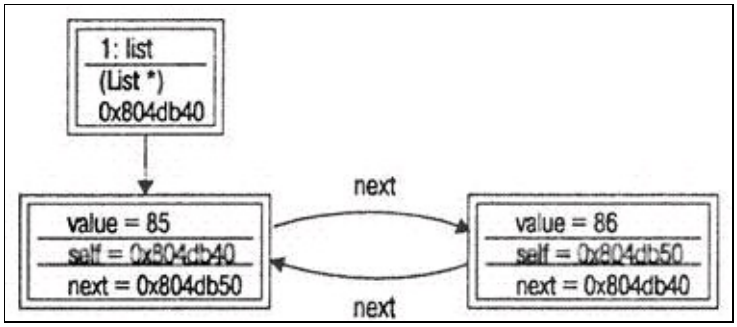
Но вы можете проникнуть в суть данных намного глубже, используя отладчик, который

позволяет визуализировать данные и все существующие отношения между ними. Существуют отладчики, которые могут представить ваши данные с высоты полета над трехмерным ландшафтом виртуальной реальности или в виде трехмерного временного графика сигналов, или же просто в виде обычных блок-схем, как показано на рисунке 3.2. По мере того как вы перемещаетесь шаг за шагом по вашей программе, рисунки, подобные этим, могут оказаться ценнее, чем тысячи слов, если ошибка, за которой вы охотились, неожиданно выпрыгивает на вас, как зверь на ловца.

Даже если отладчик имеет ограниченную поддержку визуализации данных, вы все равно можете проводить визуализацию сами – либо вручную, с карандашом и бумагой, либо с помощью внешних программ построения графиков.

В отладчике DDD имеются некоторые средства визуализации, которые распространяются бесплатно (см. [URL 19]). Интересно заметить, что отладчик DDD работает со многими языками, включая Ada, C, C++, Fortran, Java, Modula, Pascal, Perl и Python (явно ортогональная конструкция).

Рис. 3.2. Пример отладочной схемы циркулярного связанного списка. Стрелки указывают на узлы.



Трассировка

Отладчики обычно сосредоточены на состоянии программ в данный момент. В ряде случаев вам необходимо нечто большее – отследить состояние программы или структуры данных через какое-то время. Если посмотреть на трассировку стека, то можно лишь сделать вывод, как попасть в эту точку напрямую. Это не дает информации о том, что вы делали до этой последовательности обращений, что особенно важно для систем, основанных на событиях.

Операторы трассировки представляют собой небольшие диагностические сообщения, которые выводятся на экран или в файл и говорят о том, что "это здесь" и "x = 2". Это примитивная методика, сравнимая с отладчиками в стиле ИСР, но она особенно эффективна при диагностировании некоторых классов ошибок, с которыми отладчики справиться не могут. Трассировка имеет большое значение в любой системе, где время само по себе является фактором: в одновременных процессах, системах реального времени и приложениях, основанных на событиях.

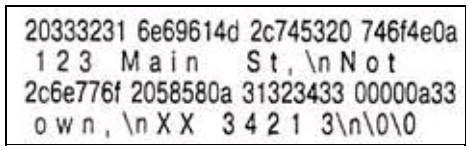
Вы можете использовать операторы трассировки для того, чтобы "вбуравиться" в текст. То есть вы можете добавлять элементы трассировки по мере продвижения вниз по дереву обращений.

Трассировочные сообщения должны быть представлены в регулярном, согласованном формате; возможно, вам захочется провести их синтаксический анализ в автоматическом режиме. Например, если вам необходимо отследить утечку ресурсов (несбалансированные

операции открытия и закрытия файлов), вы можете трассировать каждый из операторов open и close в файле журнала. Обрабатывая файл журнала с помощью программы на языке Perl, вы легко обнаружите, где встречался оператор-нарушитель open.

Искаженные переменные! Проверьте их окружение

Иногда вы исследуете переменную, ожидая увидеть небольшое целое значение, а вместо этого получаете нечто вроде 0x6e696614d. Перед тем как засучив рукава всерьез приняться за отладку, стоит посмотреть на память вокруг искаженного значения. Часто это дает вам ключ к пониманию. В данном случае, изучение окружающей памяти в символьном виде дает следующую картину:



Похоже, что кто-то указал адрес поверх счетчика цикла. Теперь, мы знаем где искать.

Рассказ о резиновом утенке

Очень простая, но весьма полезная методика поиска причины проблемы, состоит в том, чтобы разъяснить ее кому-либо. Ваш собеседник должен заглядывать через ваше плечо на экран монитора и время от времени утвердительно кивать головой (подобно резиновому утенку, ныряющему и выныривающему в ванне). Ему не нужно говорить ни слова; простое, последовательное объяснение того, что же должна делать ваша программа, часто приводит к тому, что проблема выпрыгивает из монитора и объявляет во всеуслышанье: "А вот и я!" [22].

Звучит просто, но разъясняя проблему вашему собеседнику, вы должны явно заявить о тех вещах, которые считаете само собой разумеющимися при просмотре текста вашей программы. Поскольку вам приходится озвучивать некоторые из этих положений, вы можете по-новому взглянуть на суть данной проблемы – неожиданно для самого себя.

Процесс исключения

В большинстве проектов отлаживаемая вами программа может представлять собой смесь прикладных программ, написанных лично вами и другими сотрудниками вашей проектной команды, а также программные продукты, созданные независимыми производителями (база данных, обеспечение связи, графические библиотеки, специализированные протоколы связи или алгоритмы, и т. д.) и платформенное окружение (операционная система, системные библиотеки и компиляторы).

Вероятно, ошибка кроется в операционной системе, компиляторе или продукте независимого производителя – но это не должно быть первой мыслью, приходящей вам на ум. Скорее всего, ошибка существует в тексте разрабатываемого приложения. Обычно выгоднее полагать, что прикладная программа некорректно обращается к библиотеке, нежели то, что нарушена сама библиотека. Даже если проблема заключается в продукте независимого производителя, то перед тем, как представлять отчет об ошибках, вам в любом случае надлежит исключить ошибки в вашей собственной программе.

Однажды мы работали над проектом, и старший инженер был уверен, что в системе Solaris имелось нарушение системного вызова select. Никакие убеждения или логические построения не могли изменить сложившегося у него мнения (тот факт, что все другие сетевые приложения работали прекрасно, не принимался во внимание). Неделями он составлял программы обхода этого вызова, которые, по какой-то странной причине, не способствовали решению проблемы. И когда в конце концов он был вынужден сесть за стол и прочесть документацию по вызову select, он обнаружил, в чем заключалась проблема, и исправил ее за несколько минут. Теперь мы используем выражение "вызов select нарушен" как деликатное напоминание, в тех случаях, когда один из нас начинает обвинять систему в наличии ошибки, которая, скорее всего, является его собственной.

Подсказка 26: Ищите ошибки вне пределов операционной системы

Помните: увидев следы копыт, думайте о лошадях, а не о зебрах. Скорее всего, операционная система не нарушена. Да и база данных находится в прекрасном состоянии.

Если вы "внесли всего одно изменение", и система перестала работать, то, скорее всего, именно оно, прямо или косвенно, несет ответственность за случившееся, каким бы притянутым за уши ни казалось это утверждение. Иногда то, что изменяется, находится вне вашего управления: новые версии операционной системы, компилятора, базы данных или программы независимых производителей могут вызывать проблемы и с изначально корректной программой. В ней могут обнаружиться новые ошибки. Ошибки, которые были устранены с помощью программы обхода, преодолевают действие этой программы. Если изменяются API, то изменяются и функциональные возможности; короче говоря, это уже новая история, и вам надлежит провести повторное тестирование системы в новых сложившихся условиях. Так что не спускайте глаз с графика выполнения проекта, если собираетесь провести модернизацию; может быть, придется подождать до выпуска новой версии.

Однако если вы не знаете, с чего начать, то всегда можете положиться на старый добрый двоичный поиск. Обратите внимание, не проявляются ли симптомы в одной из двух точек в тексте программы, находящихся далеко друг от друга. Затем посмотрите на точку, расположенную между ними. При наличии проблемы, ошибка «сидит» между начальной и срединной точкой; в противном случае она «сидит» между срединной и конечной точками. Продолжая действовать в этом ключе, вы сужаете область поиска, пока не выявите ошибку.

Элемент удивления

Если ошибка вызвала у вас удивление (до того, что вы еле слышно бормочете "Этого не может быть"), стоит провести переоценку истин, дорогих вашему сердцу. А все ли граничные условия вы протестировали в подпрограмме связанного списка – той, которую вы считали непробиваемой и которая, по всей вероятности, не могла стать причиной этой ошибки? А другой фрагмент текста программы, который вы использовали в течение нескольких лет, – не мог ли он все еще таить в себе ошибку?

Конечно, мог. То удивление, которое вы испытываете, когда что-то идет не так как надо, прямо пропорционально уровню доверия и веры в правильность прогоняемой программы. Поэтому, столкнувшись с «удивительным» отказом в работе программы, вы должны осознать, что одно или более ваших предположений неверны. Не приукрашивайте подпрограмму или

фрагмент текста программы, вызвавший ошибку, только потому, что «знаете», что он работает нормально. Вначале докажите это. Докажите это в реальном контексте, с реальными данными, с реальными граничными условиями.

Подсказка 27: Не предполагайте – доказывайте

Столкнувшись с удивительной ошибкой, помимо простого ее устранения, необходимо определить, а почему этот сбой не был выявлен раньше. Подумайте, не стоит ли внести поправки в модульные или иные тесты с тем, чтобы они могли выявить эту ошибку.

Кроме того, если ошибка является результатом неправильных данных, которые распространились по нескольким уровням, перед тем как вызвать взрыв, посмотрите, может быть, более усовершенствованная процедура проверки параметров в этих подпрограммах смогла бы помешать ее распространению (см. обсуждение процедур досрочного сбоя и утверждений разделе "Мертвые программы на лгут").

Пока вы собираетесь заняться этим, выясните, есть ли в программе другие фрагменты, подверженные воздействию той же ошибки? Пришло время отыскать их и устранить. Убедитесь: что бы ни произошло, вы будете знать, произойдет ли это снова.

Если устранение этой ошибки заняло много времени, спросите себя, а почему? Можете ли вы сделать что-нибудь, чтобы облегчить устранение этой ошибки в следующий раз, например, встроить усовершенствованные обработчики прерываний (для тестирования) или написать программу-анализатор файла журнала?

И наконец, если ошибка является результатом чьего-то неправильного предположения, обсудите проблему со всей командой: если имеется недопонимание со стороны одного сотрудника, то возможно, он не одинок здесь.

Проделайте все это, и наверняка в следующий раз вы будете избавлены от подобных сюрпризов.

Контрольные вопросы при отладке

• Является ли проблема прямым результатом фундаментальной ошибки или просто ее признаком?

• Ошибка действительно «сидит» в компиляторе? В операционной системе? Или в вашей собственной программе?

• Если бы вам пришлось подробно объяснить вашему коллеге, в чем состоит проблема, что бы вы ему сказали?

• Если подозрительная программа проходит модульное тестирование, то является ли оно достаточно полным? Что произойдет, если вы прогоняете модульный тест с реальными данными?

• Существуют ли условия, вызвавшие данную ошибку, где-либо еще в системе?

Другие разделы, относящиеся к данной теме:

• Программирование на основе утверждений

• Программирование в расчете на совпадение

• Вездесущая автоматизация
• Безжалостное тестирование

Вопросы для обсуждения

• Отладка сама по себе является вопросом.

19
Обработка текста

Прагматики обрабатывают тексты программ так, как столяры придают форму деревянным заготовкам. В предыдущих разделах обсуждались некоторые специфические инструментальные средства – оболочки, редакторы, отладчики – те, что мы используем в работе. Они подобны столярным долотам, ножовкам и рубанкам – инструментам, которые предназначены для выполнения одной или двух конкретных работ. Однако время от времени нам приходится выполнять некоторые преобразования, которые не могут быть осуществлены с помощью походного инструментария. В таких случаях нам необходим универсальный инструмент для обработки текста.

Языки, предназначенные для обработки текста, играют в программировании ту же роль, что станки [23] в столярном деле. Они издают шум, неуклюжи и представляют собой грубую силу. Если при работе с ними вы совершаете ошибку, то разрушенными могут оказаться целые фрагменты. Некоторые клятвенно уверяют, что этим средствам нет места в инструментарии. Но в хороших руках и станки, и языки обработки текста могут быть невероятно мощными и гибкими. Вы можете быстро придать форму материалу, делать стыки и вырезать по дереву. При надлежащем использовании эти инструменты обладают удивительной тонкостью и ловкостью. Но для овладения ими требуется время.

Число хороших языков обработки текста постоянно увеличивается. Разработчики программ для Unix часто любят использовать мощь их командных оболочек, усиленных инструментальными средствами типа awk и sed. Тем, кто предпочитает более структурированные средства, больше по душе объектно-ориентированный характер языка Python [URL 9]. Выбор некоторых падает на Tel [URL 23]. Случается, и мы предпочитаем язык Perl [URL 8] для написания коротких сценариев.

Эти языки являются важными узаконивающими технологиями. Используя их, вы можете быстро решить все проблемы с утилитами и создать прототипы идей – при работе с обычными языками на это потребовалось бы раз в пять-десять больше времени. И этот умножающий коэффициент кардинально важен для экспериментов, которые мы проводим. Потратить 30 минут на воплощение сумасшедшей идеи намного лучше, чем потратить на то же пять часов. Потратить один день на автоматизацию важных составляющих проекта – нормально, потратить неделю – может быть, и нет. В книге "The Practice of Programming" [KP99], Керниган и Пайк реализовали одну и ту же программу на пяти различных языках. Самой короткой оказалась версия на языке Perl (17 строк по сравнению со 150 строками на языке C). Работая с языком Perl, вы можете обрабатывать текст, взаимодействовать с другими программами, передавать данные по сетям, управлять web-страницами, производить арифметические действия с произвольной точностью и писать программы, которые выглядят наподобие клятвы Снупи.

Подсказка 28: Изучите язык обработки текстов

Чтобы продемонстрировать широту области применения языков обработки текста, в качестве примера мы приводим некоторые приложения, разработанные нами на протяжении последних нескольких лет:

• **Сопровождение схемы базы данных.** Набор сценариев на языке Perl обрабатывал файл с

простым текстом, содержащий определение схемы базы данных и генерировал из него:

– Инструкции SQL для создания БД
– Плоские файлы данных для заполнения словаря данных
– Библиотеки программ на языке C для доступа к БД
– Сценарии для проверки целостности БД
– Web-страницы, содержащие описания и блок-схемы БД
– XML версию схемы

• **Доступ к свойству Java.** Хорошим тоном в объектно-ориентированном программировании является ограничение доступа к свойствам объекта, что вынуждает внешние классы получать и устанавливать их через методы. Однако в общем случае, когда свойство представлено внутри класса при помощи простого поля, создание метода get и set для каждой переменной представляет собой утомительную механическую процедуру. У нас имеется сценарий Perl, который изменяет исходные файлы и вставляет правильные определения метода для всех переменных, помеченных соответствующим образом.

• **Генерирование тестовых данных.** У нас имелись десятки тысяч записей, содержащих тестовые данные, рассеянных по нескольким различным файлам разного формата, которые нуждались в соединении и преобразовании в некую форму, пригодную для загрузки в реляционную БД. Программа на Perl справилась с этим за пару часов (и в процессе обнаружила пару ошибок из-за несовместимости в исходных данных).

• **Написание книг.** Мы придаем важность тому факту, что любая программа, представленная в книге, вначале должна быть протестирована. Большинство программ, приведенных в этой книге, были протестированы. Однако, используя принцип DRY (см. "Пороки дублирования"), мы не хотели копировать и вставлять строки текста из протестированных программ в книгу. Это означало бы, что текст дублируется, поэтому велика вероятность, что мы забудем обновить пример, когда соответствующая программа изменится. В некоторых примерах нам также не хотелось утомлять вас наличием «скелета» программы, необходимым для компиляции и прогона нашего примера. Мы обратились к языку Perl. При форматировании книги вызывался относительно простой сценарий – он извлекал именованный сегмент исходного файла, выделял синтаксические конструкции и преобразовывал результат в язык, который мы используем для подготовки типографского макета книг.

• **Интерфейс между языками C и Object Pascal.** У заказчика имеется команда разработчиков, пишущих программы на языке Object Pascal, реализованном на персональных компьютерах. Требуется осуществить сопряжение их программы с телом программы, написанной на языке C. Был разработан короткий сценарий на языке Perl, который проводил синтаксический анализ файлов заголовков C, выделяя определения всех экспортированных функций и используемых ими структур данных. Затем сгенерированы модули Object Pascal с записями Pascal для всех структур C и произведен импорт определений процедур для всех функций C. Этот процесс генерирования стал частью сборки, так что при любых изменениях заголовка C происходит автоматическое конструирование нового модуля Object Pascal.

• **Генерирование интернет-документации.** Многие команды разработчиков публикуют свою документацию на внутренних интернет-сайтах. Авторами написано много программ на языке Perl, которые анализируют схемы баз данных, исходные файлы на C и C++, сборочные файлы и другие исходные тексты проекта для производства требуемой HTML-документации. Авторы также использовали язык Perl для верстки документов со стандартными верхними и нижними колонтитулами и передачи их на интернет-сайт.

Языки обработки текстов используются почти ежедневно. Многие из идей, описанных в данной книге, могут реализовываться на этих языках проще, чем на любом другом известном

языке. Эти языки облегчают написание генераторов текстов программ, которые будут рассмотрены далее.

Другие разделы, относящиеся к данной теме:

• Пороки дублирования

Упражнения

11. В вашей программе на языке C для представления одного из 100 состояний используется перечислимый тип данных. В целях отладки вам хотелось бы иметь возможность вывода состояния на печать в виде строки (в отличие от числа). Напишите сценарий, который осуществляет считывание со стандартного устройства файла следующего содержания (Ответ см. в Приложении B.):

```
name
  state_a
  state_b
   : :
```

Создайте файл name.h, содержащий следующие строки:

```
extern const char * NAME_names[]

extern const char * NAME_names[]
 typedef enum {
state_a,
state_b,
: :
} NAME;
```

а также файл name.c, содержащий следующие строки:

```
const char * NAME_names[] = {
"statea",
"state_b"
 : :
};
```

12. Дописав эту книгу до середины, авторы обнаружили, что не поместили директиву use strict во многие примеры на языке Perl. Напишите сценарий, который просматривает все файлы типа *.pl в некотором каталоге и добавляет директиву use strict в конец начального блока комментариев ко всем файлам, где это не было сделано ранее. Не забудьте сохранить резервную копию всех файлов, в которые внесены изменения. (Ответ см. в Приложении B.)

20
Генераторы текстов программ

Если столярам приходится снова и снова изготавливать одну и ту же деталь, они идут на хитрость. Они делают для себя шаблон. Если они сделают шаблон один раз, то время от времени они могут воссоздавать некоторый фрагмент работы. Шаблон избавляет столяров от излишней сложности и снижает вероятность ошибки, позволяя мастеру сосредоточиться на качестве работы.

Программисты часто оказываются в аналогичном положении. От них требуется достижения той же функциональности, но в различных контекстах. Информация должна быть воспроизведена в различных местах. А иногда, экономя на повторном наборе текста, мы просто защищаем самих себя от болей в запястье.

Подобно столяру, вкладывающему свое время в шаблон, программист может построить генератор текста. Его можно использовать всю оставшуюся жизнь проекта практически бесплатно.

Подсказка 29: Пишите текст программы, которая пишет текст программы

Существует два основных типа генераторов текста:

1. Пассивные генераторы текста запускаются один раз для достижения результата. Начиная с этого момента результат становится независимым – он отделяется от генератора текста. Мастера, обсуждаемые в разделе "Злые волшебники", вместе с некоторыми средствами CASE являются примерами пассивных генераторов текста.

2. Активные генераторы текста используются всякий раз, когда возникает необходимость в результатах их работы. Этот результат создается по принципу "выбросить и забыть" – он всегда может быть воспроизведен с помощью генератора текста. Зачастую активные генераторы считывают некоторую форму сценария или управляющего файла для получения конечного результата.

Пассивные генераторы

Пассивные генераторы текста экономят время, необходимое на набор текста. Как только результат получен, он становится полностью приспособленным для использования в качестве исходного файла в данном проекте; он должен быть отредактирован, скомпилирован и передан системе управления исходным текстом, как и любой другой файл. О его происхождении никто и не вспомнит. Пассивные генераторы текста применяются во многих случаях:

• *Создание новых исходных файлов.* Пассивный генератор текста может создавать шаблоны, директивы управления исходным текстом, сведения об авторских правах и стандартные блоки комментариев для каждого нового файла в некотором проекте. Мы настроили наши редакторы на выполнение этого действия всякий раз при создании нового файла: при редактировании новой программы на языке Java в новом буфере редактора автоматически окажутся уже заполненные блок комментариев, директива пакета и описание структурного класса.

• *Осуществление двоичных преобразований в языках программирования.* Мы начали писать эту книгу, используя систему troff, но после пятнадцатого раздела перешли на LaTeX. Мы

написали генератор текста, который считывал исходный текст из troff и преобразовывали его в формат LATJX. Точность составила 90 %; остальное мы делали вручную. Это является интересной особенностью пассивных генераторов текста: они не должны отличаться абсолютной точностью. Вы выбираете, какое усилие необходимо вложить в генератор, в сравнении с энергией, которую вы тратите на устранение ошибок в выходной информации.

• *Создание таблиц поиска и других ресурсов, вычисление которых является дорогой операцией.* Вместо того, чтобы вычислять тригонометрические функции, во многих старых графических системах использовались таблицы синусов и косинусов. Обычно эти таблицы создавались пассивным генератором текста и затем копировались в исходный текст программы.

Активные генераторы текста

Пассивные генераторы текста являются не более чем удобством, но их активные родственники являются необходимостью, если вы хотите следовать принципу DRY. С помощью активного генератора текста вы можете использовать представление некоторого фрагмента знания и преобразовать его во все формы, необходимые вашему приложению. Это не является дублированием, поскольку эти формы являются расходным материалом и создаются генератором текста по мере необходимости (отсюда термин "активный").

Когда бы вам ни приходилось организовывать совместную работу двух совершенно разных сред, стоит подумать об использовании активных генераторов текста.

Допустим, вы разрабатываете приложение БД. В этом случае вы имеете дело с двумя средами – базой данных и языком программирования, который используется для доступа к БД. У вас есть схема, и вам необходимо определить низкоуровневые конструкции, отражающие компоновку определенных таблиц БД. Вы могли бы просто запрограммировать их напрямую, но при этом нарушается принцип DRY: знание схемы было бы выражено дважды. Если схема меняется, вам необходимо помнить и о соответствующем изменении текста программы. Если из таблицы удаляется столбец, а база текста программы не меняется, то может статься, что ошибка не проявится даже при компиляции. Первый раз вы узнаете об этом во время тестирования, когда начнутся сбои (или же от пользователя).

Рис. 3.3. Активный генератор создает текст программы из схемы базы данных

Альтернативой этому является использование активного генератора текста – берется схема и используется для генерации исходного текста конструкций, как показано на рисунке 3.3. Теперь при любом изменении схемы будет происходить и автоматическое изменение программы, используемой для доступа к ней. При удалении столбца исчезает и соответствующее поле в конструкции, и любая высокоуровневая программа, использующая этот столбец, не пройдет компиляцию. Ошибку удалось заметить во время компиляции, а не в процессе сборки. Конечно, эта схема работает только в том случае, если вы сделаете генерацию текста частью самого процесса сборки [24].

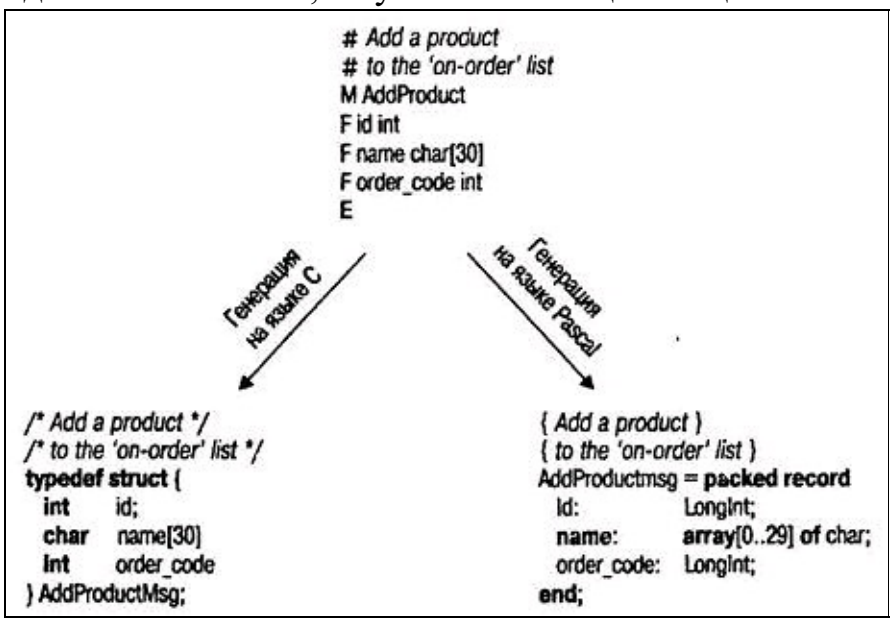
Другим примером слияния сред с помощью генераторов текста является случай, когда в одном и том же приложении использованы различные языки программирования. Для того чтобы общаться, каждой программной базе необходима некоторая общая информация – например,

структуры данных, форматы сообщений и имена полей. Вместо того, чтобы дублировать эту информацию, используйте генератор текста. В ряде случаев можно проводить синтаксический анализ информации из исходных файлов на одном языке и использовать ее для генерации текста на другом. Хотя зачастую легче выразить ее более простым, независимым от языка представлением и сгенерировать программу для обоих языков, как показано на рисунке 3.4. Также можно посмотреть ответ к упражнению 13 (см. Приложение В) в качестве примера того, как отделить синтаксический анализ представления плоского файла от генерации текста.

Генераторы текста не должны быть слишком сложными

Весь этот разговор об активном «этом» и пассивном «том», может создать у вас впечатление, что генераторы текста – сложные звери. Им не надо быть сложными. Обычно самой сложной частью является синтаксический анализатор, который обрабатывает входной файл. Не усложняйте входной формат, и генератор текста станет простым. Обратите внимание на ответ к упражнению 13 (см. Приложение В): в реальности генерация текста представляет собой в основном операторы print.

Рис. 3.4. Генерирование теиста из представления, независимого от языка. Строки во входном файле, начинающиеся с буквы M, означают начало определения сообщения, буква F означает строки с определениями полей, а буква E – конец сообщения.



Генераторы текста не всегда генерируют тексты программ

Хотя многие и примеров, приведенных в данном разделе, демонстрируют тексты программ, которые производят исходные тексты программ, на практике это не всегда так. Вы можете применять генераторы текстов для создания выходного файла в любом формате (HTML, XML, простой текст) – любого текста, который является входной информацией в какой-либо части вашего проекта.

Другие разделы, относящиеся к данной теме:

• Пороки дублирования
• Преимущество простого текста
• Злые волшебники
• Вездесущая автоматизация

Упражнения

13. Напишите генератор текста, который обрабатывает входной файл, изображенный на рисунке 3.4, и генерирует выходной файл на двух языках по вашему выбору. Попытайтесь упростить добавление новых языков. (Ответ см. в Приложении В.)

Глава 4
Прагматическая паранойя

Подсказка 30: Невозможно написать совершенную программу

Ваши чувства задеты? Не стоит принимать эту подсказку близко к сердцу. Примите ее как жизненную аксиому. Заключите ее в объятья. Восславьте ее. Поскольку совершенных программ в природе не существует. За всю краткую историю информатики никому не удалось написать ни одного совершенного фрагмента программы. Маловероятно, что вы станете первым. И когда вы примете это как существующий факт, то перестанете тратить время и энергию впустую в погоне за призрачной мечтой.

Каким же образом, учитывая эту гнетущую реальность, может прагматик обратить ее себе на пользу? Это и является темой данной главы.

Каждый знает, что лично он – лучший водитель на планете Земля. Остальному человечеству далеко до него, проезжающего под запрещающие знаки, мотающегося из ряда в ряд, не подающего сигналы поворота, разговаривающего по телефону, читающего за рулем газету и просто не живущего по общепринятым нормам. Так что мы ездим осторожно. Мы обнаруживаем неприятность до того, как она происходит, ожидаем непредвиденное и никогда не оказываемся в положении, из которого не сможем выпутаться сами.

Аналогия с написанием программ весьма очевидна. Мы постоянно взаимодействуем с программами, написанными другими людьми, программами, которые не отвечают нашим высоким требованиям, и имеем дело с входными параметрами, которые являются или не являются допустимыми. Нас учат программировать с осторожностью. При наличии любого рода сомнений мы проверяем правильность всей поступающей к нам информации. Мы используем утверждения для обнаружения неверных данных. Мы проверяем непротиворечивость, устанавливаем ограничения на столбцы базы данных и вообще высоко ценим самих себя.

Но прагматики идут дальше. Они не доверяют даже самим себе. Зная, что никто не пишет совершенных программ, включая их самих, прагматики пишут программы, защищаясь от собственных ошибок. Первый рубеж обороны описан в разделе "Проектирование по контракту": клиенты и поставщики должны договориться о правах и обязанностях.

В разделе "Мертвые программы не лгут" говорится о гарантиях того, что процедура устранения ошибок не нанесет вреда. Поэтому мы попытаемся чаще проверять нашу программу и завершать ее аварийно, если она работает не так, как надо.

В разделе "Программирование с применением утверждений" описан простой метод проверки "на ходу" – программа, которая активно проверяет ваши предположения.

Исключения, как и любая другая методика, может причинить больше вреда, чем пользы, если ее применять неправильно. Мы обсудим эти аспекты в разделе "Случаи, когда необходимо использовать исключения".

По мере того как ваши программы приобретают большую динамику, вы начинаете жонглировать системными ресурсами – памятью, файлами, устройствами и т. п. В разделе "Балансировка ресурсов" предлагаются способы того, как не ронять те предметы, которыми вы жонглируете.

Поэтому будем осторожными в этом мире несовершенных систем, устаревших временных

масштабов, смешных инструментальных средств и невыполнимых требований.

Если все общество отклоняется от нормы, чтобы понять вас, скорее всего это паранойя.

Вуди Аллен

21
Проектирование по контракту

Ничто не ошеломляет людей так сильно, как здравый смысл и честная сделка.

Ральф Уолдо Эмерсон, Эссе

Работать с компьютерными системами всегда непросто. Работать с людьми еще сложнее. И поскольку мы (как биологический вид) развиваемся достаточно долго, то у нас явно было больше времени на выяснение природы человеческих взаимоотношений. Некоторые из тех решений, к которым мы пришли в течение нескольких последних тысячелетий, могут быть применены и к созданию программного обеспечения. Одним из лучших способов, с помощью которого можно удостовериться в честности заключаемой сделки, является контракт.

В контракте определены ваши права и обязанности, а также права и обязанности другой стороны. В дополнение к этому существует соглашение, касающееся юридических последствий, возникающих в том случае, если какая-либо из сторон окажется не в состоянии соблюдать условий контракта.

Допустим, у вас есть контракт на работу, где определены количество отрабатываемых часов и правила поведения, которым вы обязаны следовать. В ответ фирма платит вам заработную плату и предоставляет другие льготы. Каждая из сторон выполняет свои обязательства, а в результате выигрывают все.

Эта идея используется во всем мире – и формальным, и неформальным образом – для того, чтобы помочь людям во взаимодействии. Можем ли мы применить этот же принцип, чтобы способствовать взаимодействию программных модулей? Ответ на этот вопрос положительный.

Проектирование по контракту

Бертран Мейер [Mey97b] разработал концепцию проектирования по контракту для языка Eiffel [25]. Это простая, но мощная методика, сосредоточенная на документировании (и согласовании) прав и обязанностей программных модулей в целях обеспечения корректности программы. Так что же означает "корректная программа"? Это та программа, которая делает не более и не менее того, на что она претендует. Документирование и подтверждение указанных претензий лежит в основе принципа проектирования по контракту (в дальнейшем, для краткости, будем называть его ППК).

Каждая функция и метод в программной системе осуществляет некоторое действие. До того как подпрограмма начнет выполнять это действие, она может иметь некие виды на состояние окружающего мира, а также может констатировать состояние окружающего мира на момент завершения работы. Б. Мейер описывает эти виды и претензии следующим образом:

• **Предусловия.** Требования подпрограммы – то, что обязано быть истинным для того, чтобы подпрограмма могла вызываться. Если предусловия нарушены, программа не должна вызываться ни в коем случае. Ответственность за передачу качественных данных лежит на вызывающей программе (см. врезку ниже "Кто несет ответственность?").

• **Постусловия.** Состояние окружающего мира на момент завершения работы подпрограммы – то, что гарантируется подпрограммой. Сам факт того, что в ней имеется

постусловие, подразумевает, что подпрограмма завершит свою работу: бесконечные циклы не допускаются.

• **Инварианты класса.** Класс гарантирует, что данное условие всегда истинно с точки зрения вызывающей программы. Во время внутренней обработки подпрограммы инвариант может и не выполняться, но к моменту выхода из подпрограммы и передачи управления обратно к вызывающей программе инвариант обязан быть истинным. (Следует заметить, что класс не может давать неограниченное право доступа для записи к любому элементу данных, участвующему в инварианте.)

Рассмотрим контракт на создание программы, которая осуществляет вставку значения данных в упорядоченный список уникальных данных. При работе с iContract (препроцессором для языка Java, который можно загрузить с [URL 17]) этот контракт может быть реализован следующим образом:

```
/**г
* ©invariant forall Node n in elements() |
*   n.prev() != null
*     implies
*       n.value().compareTo(n.prev().value()) > 0
 */
public class DBC_list {
/**
* @pre contains(aNode) == false
* @post contains(aNode) == true
*/
public void insertNode(final Node aNode) {
//…
```

Здесь стоит сказать, что элементы в списке всегда должны располагаться в порядке возрастания. При вставке нового элемента этот порядок уже может быть нарушен, и мы гарантируем, что элемент будет обнаружен после того, как он был вставлен в список.

Вы формулируете эти предусловия, постусловия и инварианты на целевом языке программирования, возможно, с некоторыми расширениями. Например, iContract предоставляет операторы логики предикатов – forall, exists и implies, дополняя обычные конструкции языка Java. Ваши утверждения могут сделать запрос о состоянии любого объекта, к которому имеется доступ со стороны метода, но удостоверьтесь, что запрос не окажет никакого побочного воздействия (см. ниже врезку "Утверждения и побочные условия").

ППК и параметры-константы

Во многих случаях постусловие будет использовать параметры, переданные в метод, для проверки правильности поведения. Но если подпрограмме разрешено изменять переданный параметр, то у вас есть возможность обойти условия контракта. В отличие от языка Java, язык Eiffel не позволяет подобных действий. В данном случае для указания наших намерений, сводящихся к неизменяемости параметра в пределах метода, используется ключевое слово final (из языка Java). Это не является "защитой от дурака" – подклассы не имеют ограничений при повторном определении параметра как не являющегося окончательным. В качестве альтернативы можно использовать синтаксис variable@pre (принятый в iContract), чтобы получить исходное значение переменной, существовавшее на момент входа в метод.

Следовательно, в контракте между подпрограммой и любой потенциально вызывающей ее программой может быть записано следующее:

> "Если вызывающая программа выполняет все предусловия подпрограммы, то подпрограмма гарантирует, что по завершении ее работы все постусловия и инварианты будут истинными".

Если одна из сторон нарушает условия контракта, то применяется предварительно согласованная мера, например, возбуждается исключение или происходит завершение работы программы. Что бы ни происходило, вы не ошибетесь, утверждая, что нарушение условий контракта есть ошибка. Это происходит далеко не всегда, и поэтому предусловия не должны использоваться для осуществления таких процедур, как проверка правильности данных, вводимых пользователем.

Подсказка 31: Проектируйте в соответствии с контрактами

В разделе «Ортогональность» рекомендуется создавать «скромные» программы. В данном случае упор делается на «ленивую» программу: проявите строгость в том, что вы принимаете до начала работы, и обещайте как можно меньше взамен. Следует помнить, что если в контракте указано, что вы принимаете все условия, а взамен обещаете весь мир, то вам придется написать… ну очень большую программу!

Наследование и полиморфизм являются краеугольными камнями объектно-ориентированных языков программирования и представляют собой область, в которой принцип программирования по контракту может проявиться особенно ярко. Предположим, что вы используете наследование при создании связи типа «это-схоже-с-тем», где один класс «схож-с-тем» (другим) классом. Вероятно, вы действуете в соответствии с принципом замещения, изложенным в книге "Liskov Substitution Principle" [Lis88]:

> "Использование подклассов должно осуществляться через интерфейс базового класса, но при этом пользователь не обязан знать, в чем состоит различие между ними".

Другими словами, вы хотите убедиться в том, что вновь созданный подтип действительно "схож-с тем" (базовым) типом – что он поддерживает те же самые методы и эти методы имеют тот же смысл. Этого можно добиться при помощи контрактов. Контракт необходимо определить единожды (в базовом классе) с тем, чтобы он применялся к вновь создаваемым подклассам автоматически. Подкласс может (необязательно) использовать более широкий диапазон входных значений или же предоставлять более жесткие гарантии. Но, по крайней мере, подкласс должен использовать тот же интервал и предоставлять те же гарантии, что и родительский класс.

Рассмотрим базовый класс Java, именуемый java.awt.Component. Вы можете обрабатывать любой визуальный элемент в AWT или Swing как тип Component и не знать, чем является подкласс в действительности – кнопкой, подложкой, меню или чем-то другим. Каждый отдельный элемент может предоставлять дополнительные, специфические функциональные возможности, но, по крайней мере, он должен предоставлять базовые средства, определенные типом Component. Однако ничто не может помешать вам создать для типа Component подтип, который предоставляет методы с правильными названиями, приводящие к неправильным

результатам. Вы легко можете создать метод paint, который ничего не закрашивает, или же метод setFont, который не устанавливает шрифт. AWT не обладает контрактами, которые способны обнаружить факт нарушения вами соглашения.

При отсутствии контракта все, на что способен компилятор, – это дать гарантию того, что подкласс соответствует определенной сигнатуре метода. Но если мы составим контракт для базового класса, то можем гарантировать, что любой будущий подкласс не сможет изменять значения наших методов. Например, вы составляете контракт для метода setFont (подобный приведенному ниже), гарантирующий, что вы получите именно тот шрифт, который установили:

```
/**
* @pre f != null
* @post getFont() == f
 */
public void setFont(final Font f) {
//…
```

Реализация принципа ППК

Самая большая польза от использования принципа ПИК состоит в том, что он ставит вопросы требований и гарантий во главу угла. В период работы над проектом простое перечисление факторов – каков диапазон входных значений, каковы граничные условия, что можно ожидать от работы подпрограммы (или, что важнее, чего от нее ожидать нельзя), – является громадным шагом вперед в написании лучших программ. Не обозначив эти позиции, вы скатываетесь к программированию в расчете на совпадение (см. раздел "Программирование в расчете на стечение обстоятельств"), на чем многие проекты начинаются, заканчиваются и терпят крах.

В языках программирования, которые не поддерживают в программах принцип ППК, на этом можно было бы и остановиться – и это неплохо. В конце концов, принцип ППК относится к методикам проектирования. Даже без автоматической проверки вы можете помещать контракт в текст программы (как комментарий) и все равно получать от этого реальную выгоду. По меньшей мере, закомментированные контракты дают вам отправную точку для поиска в случае возникновения неприятностей.

Утверждения

Документирование этих предположений уже само по себе неплохо, но вы можете извлечь из этого еще большую пользу, если заставите компилятор проверять имеющийся контракт. Отчасти вы можете эмулировать эту проверку на некоторых языках программирования, применяя так называемые утверждения (см. "Программирование утверждений"). Но почему лишь отчасти? Разве вы не можете использовать утверждения для всего того, на что способен принцип ППК?

К сожалению, ответ на этот вопрос отрицательный. Для начала, не существует средств, поддерживающих распространение действия утверждений вниз по иерархии наследования. Это означает, что если вы отменяете метод базового класса, у которого имеется свой контракт, то утверждения, реализующие этот контракт, не будут вызываться корректно (если только вы не продублируете их вручную во вновь написанной программе). Не забывайте, что прежде чем

выйти из любого метода необходимо вручную вызвать инвариант класса (и все инварианты базового класса). Основная проблема состоит в том, что контракт не соблюдается автоматически.

Кроме того, отсутствует встроенный механизм «старых» значений; т. е. значений, которые существовали на момент входа в метод. При использовании утверждений, обеспечивающих соблюдение условий контрактов, к предусловию необходимо добавить программу, позволяющую сохранить любую информацию, которую вы намерены использовать в постусловии. Сравним это с iContract, где постусловие может просто ссылаться на "variabie@pre", или с языком Eiffel, который поддерживает принцип "old expression".

И наконец, исполняющая система и библиотеки не предназначены для поддержки контрактов, так что эти вызовы не проверяются. Это является серьезным недостатком, поскольку большинство проблем обнаруживается именно на стыке между вашей программой и библиотеками, которые она использует (более детально этот вопрос обсуждается в разделе "Мертвые программы не лгут").

Поддержка ППК в языках программирования

Языки программирования, в которых имеется встроенная поддержка ППК (например, Eiffel и Sather[URL 12]) осуществляют автоматическую проверку предусловий и постусловий в компиляторе и исполняющей системе. В этом случае вы оказываетесь в самом выгодном положении, поскольку все базовые элементы программы (включая библиотеки)должны выполнять условия соответствующих контрактов.

Но как быть, если вы работаете с более популярными языками типа C, C++, и Java? Для этих языков существуют препроцессоры, которые обрабатывают контракты, инкапсулированные в первоначальный исходный текст как особые комментарии. Препроцессор разворачивает эти комментарии, преобразуя их в программу, которая контролирует утверждения.

Если вы работаете с языками C и C++, то попробуйте изучить Nana [URL 18]. Nana не осуществляет обработку наследования, но использует отладчик во время выполнения программы для отслеживания утверждений новаторским методом.

Для языка Java существует средство iContract [URL 17]. Оно обрабатывает комментарии (в формате JavaDoc) и генерирует новый исходный файл, содержащий логику утверждений.

Препроцессоры уступают встроенным средствам. Они довольно муторно интегрируются в проект, а другие используемые вами библиотеки останутся без контрактов. И тем не менее, они могут принести большую пользу; когда проблема обнаруживается подобным способом – в особенности та, которую по-другому найти просто невозможно, – это уже сродни работе волшебника.

ППК и аварийное завершение работы программы

ППК прекрасно сочетается с принципом аварийного завершения работы программы (см. "Мертвые программы не лгут"). Предположим, что есть метод, вычисляющий квадратные корни (подобный классу DOUBLE в языке Eiffel). Этот метод требует наличия предусловия, которое ограничивает область действия положительными числами. Предусловие в языке Eiffel объявляется с помощью ключевого слова require, а постусловие – с помощью ключевого слова ensure, так можно записать:

Sqrt: DOUBLE is

```
-- Подпрограмма вычисления квадратного корня
require
  sqrt_arg_must_be_positive: Current >= 0;
--- ...
 --- здесь происходит вычисление квадратного корня
 --- ...
ensure
((Result*Result) – Current).abs <= epsilon*Current.abs;
-- Результат должен находиться в пределах погрешности
end;
```

Кто несет ответственность!

Кто несет ответственность за проверку предусловия, вызывающей программы или вызываемой подпрограммы? Если эта проверка реализована как часть самого языка программирования, то никто: предусловие тестируется "за кулисами" после того, как вызывающая программа обращается к подпрограмме, но до входа в саму подпрограмму. Следовательно, если необходимо явным образом проверить параметры, это должно быть выполнено вызывающей программой, потому что подпрограмма сама некогда не сможет увидеть параметры, которые нарушают ее предусловие. (В языках без встроенной поддержки вам пришлось бы окружить вызываемую подпрограмму преамбулой и/или заключением, которые проверяют эти утверждения.)

Рассмотрим программу, которая считывает с устройства ввода номер, извлекает из него квадратный корень (вызывая функцию sqrt) и выводит результат на печать. Функция sqrt имеет предусловие – ее аргумент не должен быть отрицательным числом. Если пользователь вводит отрицательное число, то именно вызывающая программа должна гарантировать, что это число не будет передано функции sqrt. Вызывающая программа может воспользоваться многими вариантами: она может завершить работу, выдать предупреждение и начать считывать другое число, она также может преобразовать число в положительное и добавить к результату, выданному функцией Sqrt, мнимую единицу. Какой бы вариант ни использовался, эта проблема определенно не связана с функцией sqrt.

Выражая область значений функции извлечения квадратного корня в предусловии подпрограммы sqrt, вы перекладываете ответственность за правильность на вызывающую программу, которой она принадлежит. Затем вы можете спокойно продолжать разработку подпрограммы sqrt, зная, что ее входные параметры не выйдут за пределы соответствующей области.

Если ваш алгоритм извлечения квадратного корня не работает (или выходит за пределы погрешности), вы получите сообщение об ошибке и трассировку стека, указывающую на цепочку вызовов.

Если вы передаете sqrt отрицательный параметр, рабочая среда Eiffel выводит на печать ошибку "sqrt_argjnust_be_positive" (аргумент функции sqrt должен быть положительным) наряду с трассировкой стека. Этот вариант реализован лучше, чем его аналогия в языках типа Java, C, и C++, где при передаче отрицательного числа в sqrt выдается специальное значение NaN (Not a Number – не число). Далее по ходу программы, когда вы попытаетесь произвести со значением NaN некие математические действия, результаты этого будут поистине удивительными.

Проблему намного проще найти и диагностировать "не сходя с места", при аварийном завершении работы программы.

Другие случаи применения инвариантов

До этого момента мы обсуждали предусловия и постусловия, применимые к отдельным методам и инварианты, которые, в свою очередь, применимы ко всем методам в пределах класса, но есть и другие полезные способы применения инвариантов.

Инварианты цикла

Понимание граничных условий для нетривиального цикла может оказаться проблематичным. Циклы испытывают воздействие "проблемы банана" (я знаю, как записать по буквам слово «банан», но не знаю, в какой момент нужно остановиться), ошибки "постов охраны" (путаница в том, что подсчитывать: сами посты или интервалы между ними) и вездесущей ошибки завышения (занижения) [URL 52].

В подобных ситуациях инварианты могут быть полезными: инвариант цикла представляет собой оператор возможной цели цикла, но он обобщен таким образом, что также истинен перед тем, как цикл выполняется, и при каждой итерации, осуществляемой с помощью цикла. Его можно считать контрактом в миниатюре. Классическим примером является подпрограмма поиска максимального элемента в массиве.

```
int m = arr[0]; // пример предполагает, что длина массива > 0
int i = 1;
// Инвариант цикла: m = max(arr[0:i-1])
while (i < arr.length) {
   m = Math.max(m, arr[i]);
   i = i + 1;
}
```

(arr [m:n] – принятое обозначение фрагмента массива, элементы которого имеют индексы от m до n). Инвариант должен быть истинным до начала выполнения цикла, а тело цикла должно гарантировать, что инвариант будет оставаться истинным во время выполнения цикла. Таким образом, нам известно, что инвариант истинен после выполнения цикла, и следовательно наш результат является достоверным. Инварианты цикла могут быть запрограммированы в явном виде (как утверждения); они также полезны при проектировании и документировании.

Семантические инварианты

Вы можете использовать семантические инварианты для выражения неизменных требований при составлении своего рода "философского контракта".

Однажды авторы книги написали программу обработки транзакций для дебетовых банковских карт. Главное требование заключалось в том, что пользователь дебетовой карты не должен проводить на своем счете одну и ту же транзакцию. Другими словами, ошибка скорее повлечет за собой отмену обработки транзакции, чем инициирует обработку дублированной транзакции – независимо от характера сбоя в системе.

Это простое правило, исходящее непосредственно из требований, доказало свою полезность

при отсеивании сложных сценариев исправления ошибок и является руководством при детальном проектировании и реализации во многих областях.

Но убедитесь в том, что вы не смешиваете требования, представляющие собой жесткие, неизменные законы с теми, что являются не более чем политикой, которая может измениться вместе с правящим режимом. Именно поэтому мы используем термин "семантические инварианты" – он должен занимать главенствующее место при определении сути предмета и не подчиняться прихотям политики (для которой предназначаются более динамичные правила ведения бизнеса).

Если вы обнаруживаете подходящее требование, убедитесь, что оно становится неотъемлемой частью любой создаваемой вами документации – будь то маркированный список в требованиях, которые подписываются в трех экземплярах, или большое объявление на обычной лекционной доске, которое не заметит разве что слепой. Постарайтесь сформулировать его четко и однозначно. Например, в случае с дебетовой картой можно было бы записать:

ERR IN FAVOR OF THE CONSUMER (ОШИБКА В ПОЛЬЗУ КЛИЕНТА)

Это и есть четкая, сжатая, однозначная формулировка, которая применима к различным областям системы. Это наш контракт со всеми пользователями системы, наша гарантия ее поведения.

Динамические контракты и агенты

До сих пор мы говорили о контрактах как о неких фиксированных, раз и навсегда установленных спецификациях. Но в случае с автономными агентами этого быть не должно. Из определения автономных агентов следует, что они могут отвергать запросы, которые не хотят выполнять. Они могут обговаривать условия контракта – "я не могу предоставить то-то и то-то, но если вы дадите мне вот это, тогда я смогу предоставить что-то другое".

Конечно, любая система, которая полагается на технологию агентов, обладает критической зависимостью от положений контракта, даже если они генерируются динамически.

Только представьте себе: при достаточном количестве элементов и агентов, которые для достижения конечной цели могут обговаривать свои собственные контракты между собой, можно было бы просто выйти из кризисной ситуации, связанной с производительностью, позволив программам решать проблемы за нас.

Но если мы не можем использовать контракты «вручную», то мы не сможем использовать их и автоматически. Поэтому в следующий раз, когда вы будете проектировать фрагмент программы, проектируйте и его контракт.

Другие разделы, относящиеся к данной теме:

- Ортогональность
- Мертвые программы не лгут
- Программирование утверждений
- Балансировка ресурсов
- Несвязанность и закон Деметера
- Временное связывание
- Программирование в расчете на совпадение
- Программа, которую легко тестировать
- Команды прагматиков

Вопросы для обсуждения

• Информация к размышлению: Если принцип ППК является столь мощным, почему бы не применять его более широко? Насколько сложно выйти на контракт? Заставляет ли он вас думать о вещах, которые вы бы в данный момент проигнорировали? Заставляет ли он вас ДУМАТЬ? Это явно небезопасный принцип!

Упражнения

14. Из чего получается удачный контракт? Можно добавлять любые предусловия и постусловия, но есть ли от них толк? Не могут ли они принести больше вреда, чем пользы? Определите, какими являются контракты в примере ниже и упражнениях 15 и 16: удачными, неудачными, уродливыми, и объясните, почему.

Рассмотрим вначале пример, написанный на языке Eiffel. Имеется программа для добавления STRING к двунаправленному циклическому списку (следует помнить, что предусловия обозначены require, а постусловия – ensure).

```
-- Добавляем элемент в двунаправленный список,
-- и возвращаем вновь созданный узел (NODE).
add_tem (item: STRING): NODE is
require
item /= Void -- /= означает 'не равно'.
deferred -- Абстрактный базовый класс
ensure
result.next.previous = result -- Проверка связей вновь
result.previous.next = result -- вновь добавленного узла.
find_item(item) = result  -- Должен найти его.
end
```

15. Теперь рассмотрим пример на языке Java – нечто подобное примеру, из упражнения 14. Оператор InsertNumber вставляет целое число в упорядоченный список. Предусловия и постусловия обозначены в соответствии с сайтом iContract (см. [URL 17]). (Ответ см. в Приложении В.)

```
private int data[];
/**
* @post data[index-1] < data[index] &&
* data[index] == aValue
*/
public Node insertNumber (final int aValue)
{
int index = findPlaceToInsert(aValue);
...
```

16. Фрагмент стекового класса на языке Java. Можно ли назвать этот контракт удачным? (Ответ см. в Приложении В.)

```
/**
* @pre anItem != null // Требует реальных данных
```

```
* @post pop() == anltem // Проверяет их наличие
* // в стеке
*/
public void push(final String anltem)
```

17. В классических примерах использования принципа ППК (см. упражнения 14–16) реализуется абстрактный тип данных – обычно это стек, или очередь. Но немногие действительно создают подобные разновидности низкоуровневых классов.

В данном упражнении требуется спроектировать интерфейс блендера для коктейлей. Он должен основываться на web-технологии, включаться по сети Интернет и использовать технологию CORBA, но в данный момент необходим лишь интерфейс управления. Блендер имеет десять скоростей (0 означает отключение); он не должен работать вхолостую а его скорость может единовременно изменяться на одну ступень (т. е. с 0 до 1, или с 1 до 2, но не сразу с 0 до 2).

Методы указаны ниже. Добавьте соответствующие предусловия и постусловия, а также инвариант. (Ответ см. в Приложении В.)

```
int getSpeed()
void setSpeed(int x)
booolean isFull()
void fill()
void empty()
```

18. Сколько чисел содержится в ряду 0, 5, 10, 15…, 100? (Ответ см. в Приложении В.)

22
Мертвые программы не лгут

Приходилось ли вам замечать, что иногда, еще до того как вы осознаете проблему, ее признаки обнаруживают посторонние люди? То же самое применимо и к программам других разработчиков. Если в одной из наших программ что-то начинает идти не так, как надо, в ряде случаев первой это «заметит» библиотечная подпрограмма. Возможно, паразитный указатель заставил нас записать в дескриптор файла какие-то бессмысленные символы. При следующем обращении к read это будет обнаружено. Возможно, что переполнение буфера привело к уничтожению счетчика, который мы собирались использовать для определения объема назначаемой памяти. Возможно, причиной сбоя окажется malloc. Логическая ошибка в одном из нескольких миллионов операторов, находящихся в тексте перед оператором выбора, означает, что его селектор больше не примет значение 1, 2 или 3. Мы берем случай default (который является одной из причин того, почему любой оператор выбора должен иметь значение по умолчанию), мы хотим знать, в какой момент произошло невозможное).

Легко поддаться умонастроению "этого не может быть, потому что не может быть никогда". Большинство из нас создавало программы, которые не проверяют, успешно ли завершилась операция закрытия файла и правильно ли записан оператор трассировки. И все сводилось к одному (к тому, что мы и так знали) – рассматриваемая программа не откажет, если будет работать в нормальных условиях. Но мы пишем программы с осторожностью. Мы ищем инородные указатели в других частях нашей программы, очищая стек. Мы выясняем, какие версии библиотек совместного пользования загружались в действительности.

Все ошибки дают вам информацию. Вы могли внушить себе, что ошибка произойти не может, и проигнорировать эту информацию. В отличие от вас, прагматики говорят себе, что если ошибка имеет место, то произошло что-то очень скверное.

> Подсказка 32: Пусть аварийное завершение работы программы произойдет как можно раньше

Аварийное завершение не означает "отправить в корзину для мусора"

Одним из преимуществ скорейшего обнаружения проблем является то, что аварийное завершение происходит как можно раньше. И во многих случаях такое завершение программы – это лучший выход из положения. Альтернативой может быть продолжение работы, запись поврежденных данных в жизненно важную базу данных или команда стиральной машине на проведение двадцатого по счету цикла отжима.

Эта философия воплощена в языке и библиотеках Java. Когда в системе выполнения случается что-то непредвиденное, происходит возбуждение исключения RuntimeException. Если это исключение не перехвачено, оно будет двигаться на верхний уровень программы и заставит ее прекратить работу, отобразив трассировку стека.

То же самое можно реализовать и на других языках программирования. Если механизм исключения отсутствует или библиотеки не возбуждают исключения, то убедитесь в том, что можете обрабатывать ошибки самостоятельно. В языке C для этого весьма полезны

макрокоманды:

```
#define CHECK(LINE, EXPECTED) \
 {int rc = LINE; \
  if (rc!= EXPECTED) \
   ut_abort(_FILE_, _LINE_, #LINE, rc, EXPECTED); }
void ut_abort(char *file, int In, char *line, int rc, int exp) {
   fprintf(stderr, "%s line %d\n'%s': expected %d, got %d\n", file, In, line, exp, rc);
exit(1);
}
```

Тогда вы можете инкапсулировать вызовы, которые никогда подведут, с помощью строки:

```
CHECK(stat("/tmp", &stat_buff), 0);
```

Если бы это не удалось, то вы бы получили сообщение, записанное в stderr:

```
source.c line 19
"stat("/tmp", &stat_buff)' : expected 0, got -1
```

Ясно, что в ряде случаев выход из выполняющейся программы просто не уместен. Возможно, вы претендуете на ресурсы, которые не освобождены, или же вам необходимо записать сообщения в журнал, завершить открытые транзакции или взаимодействовать с другими процессами. Здесь будут полезны методики, обсуждаемые в разделе "Случаи, когда необходимо использовать исключения". Однако основной принцип остается тем же – если ваша программа обнаруживает, что произошло событие, которое считалось невозможным, программа теряет жизнеспособность. Начиная с этого момента, все действия, совершаемые программой, попадают под подозрение, так что выполнение программы необходимо прервать как можно быстрее. В большинстве случаев мертвая программа приносит намного меньше вреда, чем испорченная.

Другие разделы, относящиеся к данной теме:

- Проектирование по контракту
- Когда использовать исключения

23
Программирование утверждений

В самобичевании есть своего рода сладострастие. И когда мы сами себя виним, мы чувствуем, что никто другой не вправе более винить нас.

Оскар Уайльд, Портрет Дориана Грея

В самом начале своей профессиональной карьеры каждый программист обязан выучить некую мантру. Она представляет собой фундаментальную основу компьютерных вычислений, основное вероучение, которое мы учимся применять к требованиям, конструкциям, самим программам, комментариям – словом, всему, что мы делаем. Она звучит так:

"Этого никогда не случится…"

И далее: "Через 30 лет эта программа использоваться не будет, так что для обозначения года хватит и двух разрядов". "Нужна ли интернационализация, если это приложение не будет использоваться за рубежом?" "Счетчик не может принимать отрицательное значение". "Этот оператор printf не дает сбоев".

Не стоит заниматься подобного рода самообманом, особенно при написании программ.

Подсказка 33: Если что-либо не может произойти, воспользуйтесь утверждениями, которые гарантируют, что это не произойдет вовсе

Всякий раз, когда вы начинаете думать "Ну конечно, такого просто не может произойти", проверяйте это высказывание с помощью программы. Самый простой способ осуществить это – использовать утверждения. В большинстве реализаций языков C и C++ имеется некоторая разновидность макроса assert или _assert, который осуществляет проверку логического условия. Эти макрокоманды могут представлять огромную ценность. Если указатель, передаваемый к вашей процедуре, ни в коем случае не должен принимать значение NULL, то проверьте выполнение этого условия:

```
void writeString(char *string) {
  assert(string != NULL);
...
```

Утверждения представляют собой весьма полезное средство проверки работы алгоритма. Например, вы написали умный алгоритм сортировки. Проверьте, работает ли он:

```
For (int i=0; i<num_entries-1; i++) {
  assert(sorted[i] <= sorted[i+1]);
}
```

Конечно, условие, переданное утверждению, не должно оказывать побочного воздействия (см. врезку "Утверждения и побочные эффекты"). Необходимо также помнить, что утверждения могут отключаться во время компиляции – не помещайте в макрос assert программу, которая должна быть выполнена. Утверждения не должны использоваться вместо реальной обработки ошибок. Они лишь осуществляют проверку того, что никогда не должно произойти; вы же не хотите писать программу, подобную приведенной ниже:

```
printf("Enter 'Y' or 'N': ");
ch = getchar()
assert((ch=='Y')||(ch=='N')); /* дурной тон! */
```

И поскольку имеющаяся макрокоманда assert вызывает exit, если утверждение ложно, нет никаких оснований для того, чтобы этого не могли сделать создаваемые вами версии программы. Если вам приходится освобождать ресурсы, сделайте так, чтобы невыполнение утверждения возбуждало исключение или осуществляло переход longjump к точке выхода, или же вызывало обработчик ошибки. Убедитесь в том, что программа, которая исполняется в течение миллисекунд, не использует информацию, которая привела к невыполнению утверждений.

Не отключайте утверждения

Существует расхожее недопонимание утверждений, которое провозгласили те, кто разрабатывает компиляторы и языковые среды. Оно формулируется примерно так:

> "Утверждения являются лишним бременем для программы. Поскольку они проверяют то, что никогда не должно случиться, их действие инициируется только ошибкой в тексте программы. Как только программа проверена и отправлена заказчику, необходимость в них отпадает и их надо отключить для ускорения работы программы. Утверждения нужны лишь во время отладки" .

В этом высказывании имеется два явно неправильных предположения. Первое – авторы высказывания полагают, что при тестировании обнаруживаются все ошибки. В действительности маловероятно, что процедура тестирования любой сложной программы всегда будет происходить по единому сценарию, даже при минимальном проценте перестановок в тексте программы (см. "Безжалостное тестирование"). Второе – эти оптимисты забывают, что ваша программа выполняется в опасном мире. Весьма вероятно, что во время тестирования крысы не будут прогрызать кабели, никто не будет забивать память, запуская игрушку, а файлы журналов не переполнят жесткий диск. Все это может происходить, если ваша программа выполняется в реальных условиях. Ваш первый оборонительный рубеж – проверка наличия любой вероятной ошибки, а второй – использование утверждений для обнаружения тех ошибок, которые прошли первый рубеж.

Отключение утверждений при доставке программы заказчику сродни хождению по канату без страховочной сетки на том основании, что когда-то вы уже так делали. Сумма страховки велика, но получить ее в случае падения довольно сложно.

Даже при наличии некоторых проблем с производительностью, отключите только те утверждения, которые действительно оказывают серьезное воздействие. Пример с программой сортировки, представленный выше, может быть самой важной частью вашего приложения и, наверное, должен работать быстро. Добавление процедуры проверки означает новое считывание данных, что может быть неприемлемо. Сделайте эту конкретную процедуру проверки необязательной [26], но оставьте в покое все остальные.

Утверждения и побочные эффекты

Становится как-то неловко, если программа, добавляемая для обнаружения ошибок, в

результате создает новые. Это может происходить с утверждениями в том случае, если вычисление условия имеет побочные эффекты. Например, было бы дурным тоном написать на языке Java нечто вроде:

```
while (iter.hasMoreElements() {
  Test.ASSERT(iter.nextElement() != null);
  Object obj = iter.nextElement();
// ...
}
```

Вызов .nextElement() в ASSERT обладает побочным эффектом, заключающимся в перемещении указателя цикла за выбираемый элемент, так что цикл обрабатывает лишь половину элементов совокупности. Лучше было бы записать:

```
while (iter.hasMoreElements()) {
Object obj = iter.nextElement();
Test.ASSERT(obj != null);
//…
}
```

Эта проблема являются разновидностью так называемого «Heisen-bug» – процесса отладки, изменяющего поведение отлаживаемой системы (см. [URL 52]).

Другие разделы, относящиеся к данной теме:

- Отладка
- Проектирование по контракту
- Балансировка ресурсов
- Программирование в расчете на совпадение

Упражнения

19. Быстрый тест на ощущение реальности. Какие из перечисленных «невозможных» событий могут случаться в реальности? (Ответ см. в Приложении В.)

1. Месяц, количество дней в котором меньше 28
2. Stat(".",&sb)== –1 (т. е. невозможно обращение к текущему каталогу)
3. В языке C++: a = 2; b = 3; if (a + b != 5) exit(1);
4. Треугольник, сумма величин внутренних углов которого не равна 180°
5. Минута, состоящая не из 60 секунд
6. В языке Java: (a + 1) <= a

20. Разработайте несложный класс Java для проверки утверждений (Ответ см. в Приложении В.)

24
Случаи, в которых используются исключения

В разделе "Мертвые программы не лгут" высказано предложение считать хорошим тоном проверку всех возможных ошибок, в особенности возникающих неожиданно. Однако на практике это может привести к тому, что программа станет уродливой; нормальная логика вашей программы может сойти на нет из-за ее затуманивания процедурами обработки ошибок, особенно если вы являетесь приверженцем школы программирования, девиз которой звучит так: "В подпрограмме должен иметься один-единственный оператор return" (авторы не являются приверженцами указанной школы). Нам приходилось видеть текст программы, выглядевший следующим образом:

```
retcode = OK;
if (socket.read(name)!=OK) {
  retcode = BAD_READ;
}
else
  processName(name);
  if (socket.read(address)!=OK) {
    retcode = BAD READ;
  }
  else {
    processAddress(address);
    if (socket.read(telNo)!= OK) {
      retcode= BAD_READ
   }
    else {
// etc, etc…
   }
  }
}
return retcode;
```

Но если язык программирования (по счастливой случайности) поддерживает исключения, то эту программу можно написать намного изящнее:

```
retcode = OK;
try {
  socket.read(name);
  process(name);
  socket.read(address);
  processAddress(address);
  socket.read(telNo);
// etc, etc…
}
catch (lOException e) {
   retcode = BAD_READ;
   Logger.log("Error reading individual;" +e.getMessage());
 }
```

return retcode;

Теперь схема управления отличается ясностью – вся обработка ошибок сосредоточена в одном-единственном месте.

Что является исключительным?

Одна из проблем, связанных с исключениями, заключается в том, что необходимо знать, когда их можно использовать. Мы полагаем, что не стоит злоупотреблять исключениями для нормального хода выполнения программы; они должны быть зарезервированы для нештатных ситуаций. Предположите, что неперехваченное исключение прекратит работу вашей программы, и спросите себя: "А будет ли эта программа исполняться, если удалить все обработчики исключений?". Если ответ отрицательный, то вполне возможно, что исключения используются в обстоятельствах, их не требующих.

Например, если ваша программа пытается открыть файл для чтения, а этот файл не существует, нужно ли возбуждать исключение?

Мы отвечаем следующим образом: "Это зависит от конкретного случая". Если файл должен был там находиться, то исключение гарантировано. Произошло что-то непредвиденное – файл, который вы считали существующим, похоже, исчез. С другой стороны, если вы понятия не имеете, должен ли этот файл существовать или нет, то его отсутствие уже не кажется столь исключительным и возвращение ошибки вполне уместно.

Рассмотрим пример с первой ситуацией. Представленная ниже программа открывает файл /etc/passwd, который обязан существовать во всех системах Unix. Если файл не открывается, происходит передача исключения FileNotFoundException к вызывающей программе.

```
public void open_passwd() throws FileNotFoundException {
//При этом может возбуждаться FileNotFoundException…
  ipstream = new FileInputStream("/efc/passwd");
  //…
}
```

Однако во второй ситуации может происходить открытие файла, указанного пользователем в командной строке. Здесь возбуждение исключения не гарантируется, и программа выглядит по-другому:

```
public boolean open_user_file(String name)
  throws FileNotFoundException {
  File f = new File(name);
  if (!f.exists()) {
    return false;
  }
  ipstream = new FileInputStream(f);
  return true;
}
```

Обратите внимание на то, что вызов FileInputStream все еще способен генерировать исключение, передаваемое подпрограммой. Но при этом исключение будет сгенерировано лишь в действительно исключительных обстоятельствах; простая попытка открыть несуществующий файл сгенерирует обычное возвращение ошибки.

Подсказка 34: Пользуйтесь исключениями только в исключительных случаях

Почему мы предлагаем именно такой подход к исключениям? Конечно, исключение представляет собой мгновенную нелокальную передачу управления – своего рода многоуровневый оператор goto. Программы, использующие исключения в своей обычной работе, испытывают те же проблемы с удобочитаемостью и сопровождением, которые свойственны классическим неструктурированным программам. Эти программы нарушают инкапсуляцию: подпрограммы и их вызывающие программы отличаются более сильной связанностью за счет обработки исключений.

Обработчики ошибок как альтернатива исключению

Обработчик ошибок представляет собой подпрограмму, которая вызывается при обнаружении ошибки. Вы можете зарегистрировать подпрограмму для обработки определенной категории ошибок. При возникновении одной из этих ошибок происходит вызов обработчика.

Возникают ситуации, когда вам захочется применить обработчики ошибок вместо исключений или же одновременно с ними. Понятно, что, если вы работаете с языком C, не поддерживающим исключения, это один из нескольких возможных вариантов (см. Вопрос для обсуждения ниже). Но иногда обработчики ошибки могут использоваться даже в языках (типа Java), обладающих хорошей встроенной схемой обработки исключений.

Рассмотрим реализацию приложения «клиент-сервер» с использованием средства RMI (удаленный вызов метода) в языке Java. Поскольку RMI реализован определенным способом, каждое обращение к удаленной подпрограмме должно быть подготовлено, с тем чтобы обработать ситуацию RemoteException. Добавление программы обработки этих исключений может представлять собой утомительную процедуру и означает сложность написания программы, которая могла бы работать как с локальными, так и с удаленными подпрограммами. Обойти эту трудность возможно путем инкапсулирования удаленных объектов в класс, не являющийся удаленным. Тогда этот класс сможет реализовать интерфейс обработчика ошибок, позволяя программе клиента регистрировать подпрограмму, обращение к. которой происходит при обнаружении удаленной исключительной ситуации.

Другие разделы, относящиеся к данной теме:

• Мертвые программы не лгут

Вопросы для обсуждения

• В языках программирования, не поддерживающих исключительные ситуации, часто используется иной (нелокальный) способ передачи механизма управления (например, в языке C существует средство longjmp/setjmp). Подумайте, как можно реализовать некий «суррогатный» механизм исключения, используя указанные средства. В чем состоят преимущества и опасности? Какие специальные меры необходимо предпринять для гарантии того, что эти ресурсы не «осиротеют»? Есть ли смысл использовать подобное решение всякий раз, когда вы пишете программу на языке C?

Упражнения

21. При проектировании нового класса контейнера имеются три возможных состояния ошибки:

1. Не хватает памяти для нового элемента в подпрограмме add.
2. В подпрограмме fetch не обнаружена запрашиваемая точка входа.
3. Указатель null передан к подпрограмме add.

Каким образом необходимо обрабатывать каждую из этих ошибок? Нужна ли генерация ошибки, возбуждение исключительной ситуации, или же это состояние должно игнорироваться? (Ответ см. в Приложении В).

25
Балансировка ресурсов

"Я привел тебя в этот мир", – сказал бы мой отец, – "я же могу и отправить тебя обратно. Мне это без разницы. Я сделаю еще одного такого, как ты".

Билл Косби, Отцовство

При написании программ всем нам приходится управлять ресурсами: памятью, транзакциями, потоками, файлами, таймерами – словом, всеми типами объектов, доступность которых ограничена. Большую часть времени использование ресурса следует предсказуемой схеме: ресурс назначается, используется, а затем освобождается.

Однако многие разработчики не имеют сколько-нибудь завершенного плана, касающегося распределения и освобождения ресурсов. Поэтому предлагается простая подсказка:

Подсказка 35: Доводите до конца то, что начинаете

В большинстве случаев эту подсказку применить несложно. Она всего лишь означает, что подпрограмма или объект, который назначает ресурс, должен нести ответственность за освобождение этого ресурса. Использование указанной подсказки можно рассмотреть на примере неудачной программы – приложения, открывающего файл, считывающего из него информацию о заказчике, обновляющего поле и записывающего результат обратно в файл. Чтобы сделать пример более наглядным, мы опустили процедуру обработки ошибок.

```
void readCustomer(const char *fName, Customer *cRec) {
cFiie = fopen(fName, "r+");
fread(cRec, sizeof(*cRec), 1, cFile);
}
void writeCustomer(Customer *cRec) {
rewind(cFile);
fwrite(cRec, sizeof(*cRec), 1; cFile);
fclose(cFile);
}
void updateCustomer(const char *fName, double newBalance) {
Customer cRec;
readCustomer(fName, &cRec);
cRec.balance = newBalance;
writeCustomer(&cRec);
}
```

На первый взгляд, подпрограмма updateCustomer выглядит довольно прилично. Похоже, что она реализует нужную нам логику – считывает запись, обновляет баланс и осуществляет запись обратно в файл. Однако, за внешним приличием и скрывается главная проблема. Подпрограммы readCustomer и writeCustomer тесно связаны между собой [27] – они совместно используют глобальную переменную cFile. Подпрограмма readCustomer открывает файл и сохраняет

указатель файла в переменной cFile, а подпрограмма writeCustomer использует сохраненный указатель для закрытия файла по окончании работы. Эта глобальная переменная даже не появляется в подпрограмме updateCustomer.

Почему эту программу можно считать неудачной? Представим несчастную даму-программистку из службы сопровождения, которой сказали, что спецификация изменилась – баланс должен обновляться только в том случае, если его новое значение неотрицательное. Дама открывает файл с исходным текстом и изменяет подпрограмму updateCustomer:

```
void updateCustomer(const char *fName, double newBalance) {
Customer cRec;
readCustomer(fName, &cRec);
if (newBalance >= 0.0) {
  cRec.balance = newBalance;
  writeCustomer(&cRec);
 }
}
```

Во время испытания все выглядит замечательно. Однако когда программа вводится в эксплуатацию, она «падает» через несколько часов, выдавая сообщение "Слишком много открытых файлов". Поскольку подпрограмма writeBalance в ряде случаев не вызывается, то и файл не закрывается.

У этой проблемы есть весьма неудачное решение – использовать специальный оператор выбора в подпрограмме updateCustomer:

```
void updateCustomer(const char *fName, double newBalance) {
Customer cRec;
readCustomer(fName, &cRec);
if (newBalance»= 0.0) {
 cRec.balance = newBalance;
 writeCustomer(&cRec);
}
else
fclose(cFile);
}
```

Это устраняет проблему – файл закроется независимо от нового значения баланса, но эта процедура означает, что теперь связанными оказываются три подпрограммы (через глобальную переменную cFile). Мы попадаем в ловушку, и если мы продолжаем действовать в том же духе, то все полетит под откос.

Подсказка "Доводите до конца то, что начинаете" говорит нам о том, что в идеале подпрограмма, которая назначает ресурс, обязана его и освобождать. Мы можем применить ее в данном случае, осуществляя небольшую реорганизацию программы:

```
void readCustomer(FILE *cFile, Customer *cRec) {
fread(cRec, sizeof(*cRec), 1, cFile);
}
void writeCustomer(FILE *cFile, Customer *cRec) {
rewind(cFile);
fwrite(cRec, sizeof(*cRec), 1, cFile);
}
void updateCustomer(const char *fName, double newBalance) {
FILE *cFile;
```

```
Customer cRec;
cFile = fopen(fName, "r+"); // ->>>
readCustomer(cFile, &cRec); //
if (newBalance >= 0.0) {  //
  cRec.balance = newBalance; //
  writeCustomer(cFile, &cRec); //
  } //
 Fclose(cFile);   // <<<-
}
```

Теперь всю ответственность за файл несет подпрограмма updateCustomer. Она открывает файл и (заканчивая то, что уже начала) закрывает его перед выходом. Подпрограмма осуществляет балансировку использования файла: открытие и закрытие происходят на одном и том же месте, и очевидно, что каждой процедуре открытия будет соответствовать процедура закрытия. Реорганизация также позволяет избавиться от уродливой глобальной переменной.

Вложенное роспределение ресурса

Базовая схема распределения ресурсов может быть распространена на подпрограммы, которым одновременно необходимо более одного ресурса. Поэтому есть еще два предложения:

1. Освобождайте ресурсы в последовательности, обратной той, в которой происходило их распределение. При этом можно избежать появления «осиротевших» ресурсов, если один из них содержит ссылки на другой.

2. При распределении одного и того же набора ресурсов в различных местах программы необходимо осуществлять эту операцию в одном и том же порядке. Это уменьшает вероятность взаимоблокировки. (Если процесс А требует resource1 и собирается затребовать resource2, тогда как процесс В затребовал resource2 и пытается заполучить resource1, то два процесса окажутся в состоянии вечного ожидания.)

Неважно, какой тип ресурсов используется, – транзакции, память, файлы, потоки, окна, к ним применима общая схема: кто бы ни назначал ресурс, он обязан нести ответственность за его освобождение. Однако эта концепция может получить дальнейшее развитие при работе с рядом языков программирования.

Объекты и исключения

Равновесие между распределениями ресурсов и их освобождениями напоминает о равновесии конструктора и деструктора класса. Класс представляет ресурс, конструктор создает конкретный для этого типа ресурса объект, а деструктор удаляет его из вашей области действия.

Если вы программируете на объектно-ориентированном языке, то упаковка ресурсов в классы может принести пользу. Всякий раз, когда вам необходим конкретный тип ресурса, вы создаете экземпляр объекта указанного класса. Если объект выходит из области действия или повторно запрашивается сборщиком мусора, то деструктор объекта освобождает инкапсулированный ресурс.

Этот подход обладает определенными преимуществами при работе с языками программирования типа C++, где исключения могут входить в противоречие с освобождением ресурсов.

Балансировка и исключения

Языки, поддерживающие исключения, могут сделать процедуру освобождения ресурса нетривиальной. Как удостовериться, что все ресурсы, назначенные до возбуждения исключения, освобождены надлежащим образом? В некоторой степени ответ зависит от языка программирования.

Балансировка ресурсов в исключениях языка C++

Язык C++ поддерживает механизм исключений типа try…catch. К сожалению, это означает, что всегда существует по крайней мере два возможных варианта выхода из подпрограммы, которая перехватывает, а затем повторно возбуждает исключение:

```
void doSomething(void) {
  Node *n = new Node;
  try {
   // do something
  }
  catch (…) {
    delete n;
    thow;
  }
delete n;
}
```

Заметим, что созданный нами узел освобождается дважды – один раз во время нормального выхода из подпрограммы, а второй раз в обработчике исключений. Это явное нарушение принципа DRY и проблема в сопровождении, которая может возникнуть в любой момент.

Однако в наших интересах воспользоваться семантикой языка C++. Локальные объекты автоматически разрушаются при выходе из блока, в котором они находятся. Это дает нам несколько вариантов. Если обстоятельства позволяют, можно поменять n: оно обозначает не указатель, а реальный объект Node в стеке:

```
void doSomething1(void) {
Node n;
try {
// делаем что-либо
}
catch (…) {
throw;
 }
}
```

В этом случае мы используем C++ для автоматического разрушения объекта Node независимо от того, возбуждено исключение или нет.

В случае, если замена указателя на объект невозможна, тот же самый эффект достигается при инкапсулировании ресурса (речь идет об указателе Node) в пределах другого класса.

```
// Класс оболочки для ресурсов Node
class NodeResource {
```

```
  Node *n;
public:
  NodeResource() {n = new Node;}
 ~NodeResource() {delete n;}
 Node *operator ->() {return n;}
};
void doSomething2(void) {
NodeResource n;
try {
// do something
}
catch (…) {
   throw;
 }
}
```

Теперь класс-оболочка NodeResource выступает гарантом того, что при разрушении его объектов происходит и разрушение соответствующих узлов. Для удобства класс оболочка предоставляет оператор разыменования – », с тем чтобы пользователи могли обращаться к полям в инкапсулированном объекте Node напрямую.

Поскольку эта методика столь полезна, в стандартной библиотеке C++ имеется шаблонный класс autOjDtr, обеспечивающий автоматические оболочки для динамически размещаемых объектов.

```
void doSomething3(void) {
   auto_ptr <Node> p (new Node);
// Обращение к узлу Node как p-»…
// В конце узел автоматически удаляется
}
```

Балансировка ресурсов в языке Java

В отличие от C++ язык Java реализует «ленивую» форму автоматического разрушения объекта. Объекты, ссылки на которые отсутствуют, считаются кандидатами на попадание в «мусор», и их метод finalize будет вызываться в любой момент, когда процедура сборки мусора будет претендовать на эти объекты. Представляя собой удобство для разработчиков, которым больше не приходится жаловаться на утечки памяти, в то же время он усложняет реализацию процедуры очистки ресурсов по схеме C + +. К счастью, разработчики языка Java глубокомысленно ввели компенсирующую языковую функцию – предложение finally. Если блок try содержит предложение finally, то часть программы, относящаяся к этому предложению, гарантированно исполняется только в том случае, если исполняется любая инструкция в блоке try. Неважно, возбуждается при этом исключение или нет (даже при выполнении оператора return программой в блоке try) – программа, относящаяся к предложению finally, будет выполнена. Это означает, что использование ресурса может быть сбалансировано с помощью программы типа:

```
public void doSomething() throws IOException {
File tmpFile = new File(tmpFileName);
FileWriter tmp = new FileWriter(tmpFile);
```

```
try {
 // do some work
}
finally {
 tmpFile.delete();
}
}
```

Подпрограмма использует промежуточный файл, который мы хотим удалить, независимо от того, как подпрограмма заканчивает свою работу. Блок finally позволяет нам выразить это в сжатой форме.

Случаи, при которых балансировка ресурсов невозможна

Возникают моменты, когда основная схема распределения ресурсов просто не годится. Обычно это происходит в программах, которые используют динамические структуры данных. Одна подпрограмма выделяет область в памяти и связывает ее в структуру большего размера, где она и находится в течение некоторого времени.

Хитрость здесь состоит в установлении семантического инварианта для выделения памяти. Необходимо решить, кто несет ответственность за данные в составной структуре. Что произойдет при освобождении структуры верхнего уровня? Есть три основных варианта развития событий:

1. Структура верхнего уровня также несет ответственность за освобождение любых входящих в нее подструктур. Затем эти структуры рекурсивно удалят данные, содержащиеся в них, и т. д.

2. Структура верхнего уровня просто освобождается. Любые структуры, на которые она указывает (и на которых нет других ссылок), становятся "осиротевшими".

3. Структура верхнего уровня отказывается освобождать себя, если в нее входят какие-либо подструктуры.

В этом случае выбор зависит от условий, в которых находится каждая взятая в отдельности структура данных. Однако этот выбор должен быть явным для каждого случая, и ваше решение должно реализовываться последовательно. Реализация любого из представленных вариантов на процедурном языке программирования типа C может представлять проблему: структуры данных сами по себе не являются активными. В этих условиях для каждой из основных структур предпочтительнее написать модуль, обеспечивающий стандартные средства распределения и освобождения. (Этот модуль также обеспечивает распечатку результатов отладки, преобразование в последовательную и параллельную формы и средства обхода.)

И наконец, если отслеживание ресурсов становится слишком хитрой процедурой, можно создать собственную форму ограниченной автоматической сборки «мусора», реализуя схему подсчета ссылок для ваших динамически распределенных объектов. В книге "More Effective C++" ([Меу9б]) этой теме посвящен целый раздел.

Проверка баланса

Поскольку прагматики не доверяют никому, включая авторов книги, то мы полагаем, что во всех случаях неплохо было бы написать такую программу, которая осуществляла бы реальную проверку того, освобождены ли ресурсы надлежащим образом. Для большинства приложений

это обычно означает создание оболочек для каждого типа ресурса и их использование для отслеживания всех распределений и освобождений. В некоторых точках программы логика диктует, что ресурсы находятся в определенном состоянии; для проверки этого и необходимо использовать оболочки.

Например, в программе, выполняемой на протяжении длительного времени и обслуживающей запросы, наверняка есть одна-единственная точка в начале основного цикла обработки, в которой происходит ожидание прихода следующего запроса. Именно в этой точке можно получить подтверждение тому, что с момента последнего выполнения цикла использование ресурсов не увеличилось.

При работе на более низком (но не менее полезном) уровне можно потратиться на инструментальные средства, которые (помимо всего прочего) проверяют выполняемые программы на наличие утечек памяти (регулярного неосвобождения области памяти). Весьма популярными являются Purify (www.rational.com) и Insure++ (www.parasoft.com).

Другие разделы, относящиеся к данной теме:

- Проектирование по контракту
- Программирование утверждений
- Несвязанность и закон Деметера

Вопросы для обсуждения

- Несмотря на то, что не существует надежных способов удостовериться в том, что вы освободили ресурсы, в этом могут помочь некоторые технологии проектирования, если их применять последовательно. В данной главе обсуждалось, как установить семантический инвариант, с тем чтобы основные структуры данных могли управлять освобождением памяти. Подумайте, как с помощью принципа "Проектирование по контракту" можно было бы усовершенствовать эту идею.

Упражнения

22. Некоторые разработчики программ на C и C++ обращают особое внимание на необходимость установки указателя в NULL после освобождения области памяти, на которую он ссылается. Почему это можно считать удачной идеей? (Ответ см. в Приложении B.)

23. Некоторые разработчики программ на языке Java обращают особое внимание на необходимость установки объектной переменной в NULL после окончания использования объекта. Почему это можно считать удачной идеей? (Ответ см. в Приложении B.)

Глава 5
Гибкость против хрупкости

Жизнь не стоит не месте.

Не могут стоять на месте и программы, которые мы пишем. Чтобы не отставать от сегодняшнего, близкого к кошмару, темпа изменений, необходимо приложить все усилия для написания программ слабосвязанных и гибких, насколько это возможно. В противном случае мы придем к тому, что наша программа быстро устареет или станет слишком хрупкой, что не позволит устранять ошибки, и может в конечном итоге оказаться в хвосте сумасшедшей гонки в будущее.

В разделе «Обратимость» говорится об опасностях необратимых решений. Мы расскажем вам, как принимать обратимые решения так, чтобы ваша программа смогла остаться гибкой и адаптируемой перед лицом нашего неопределенного мира.

В начале необходимо рассмотреть связывание – взаимозависимость между модулями программы. В разделе "Несвязанность и закон Деметера" будет показано, как сохранить отдельные концепции и уменьшить связывание.

Хороший способ сохранить гибкость – это писать программы меньшего размера. Изменение кода открывает перед вами возможность внесения новых дефектов. В разделе «Метапрограммирование» объясняется, как полностью вывести из текста программы подробности в то место, где их можно изменить безопаснее и проще.

В разделе "Временное связывание" рассматриваются два временных аспекта применительно к связыванию. Зависите ли вы от того обстоятельства, что «тик» наступает раньше, чем «так»? Если вы хотите сохранить гибкость, то нет!

Ключевым принципом в создании гибкой программы является отделение модели данных от их визуального представления, или воспроизведения. Несвязанность модели и ее визуального представления описана в разделе "Всего лишь визуальное представление".

И наконец, существует методика несвязанности модулей в еще большей степени за счет предоставления "места встречи", где модули могут обмениваться данными анонимно и асинхронно. Эта тема освещена в разделе "Доски объявлений".

Взяв эти методики на вооружение, вы можете написать программу, которая будет энергично вращаться – как в рок-н-ролле.

26
Несвязанность и закон Деметера

Хорошая изгородь – добрые соседи.

Роберт Фрост, Подготовка к выборам

В разделах «Ортогональность» и "Проектирование по контракту" мы высказали предположение, что выгодно писать «скромные» программы. Но эта «скромность» работает в двух направлениях: не раскрывайте себя перед другими и не общайтесь со слишком многими людьми.

Шпионы, диссиденты, революционеры и им подобные часто организованы в небольшие группы, называемые ячейками. Хотя отдельные личности в каждой ячейке могут знать друг о друге, они не знают ничего об участниках других ячеек. Если одна ячейка раскрыта, то никакое количество "сыворотки правды" неспособно выбить из ее участников информацию об их сподвижниках вне пределов ячейки. Устранение взаимодействий между ячейками убережет всех.

Мы полагаем, что этот принцип хорошо бы применить и к написанию программ. Разбейте вашу программу на ячейки (модули) и ограничьте взаимодействие между ними. Если один модуль находится под угрозой и должен быть заменен, то другие модули должны быть способны продолжить работу.

Сведение связанности к минимуму

Что произойдет, если появятся модули, которые знают друг о друге. В принципе ничего – вы не должны впадать в паранойю, как шпионы или диссиденты. Однако, необходимо внимательно следить за тем, со сколькими другими модулями вы взаимодействуете. Это важнее, чем то, каким образом вы пришли к взаимодействию с ними.

Предположим, вы занимаетесь перепланировкой своего дома или строите дом с нуля. Обычная организация включает "генерального подрядчика". Вы нанимаете подрядчика для выполнения работ, но подрядчик выполняет или не выполняет эти работы сам; работа может быть предложена разнообразным субподрядчикам. Но, будучи клиентом, вы не имеете дело с субподрядчиками напрямую, генеральный подрядчик берет от вашего имени эту головную боль на себя.

Нам бы хотелось воспользоваться той же моделью в программном обеспечении. Когда мы запрашиваем у объекта определенную услугу, то мы хотим, что бы эта услуга оказывалась от нашего имени. Мы не хотим, чтобы данный объект предоставлял нам еще какой-то объект, подготовленный третьей стороной, с которым нам придется иметь дело для получения необходимой услуги.

Предположим, что вы пишете класс, генерирующий график по данным научного прибора. Научные приборы рассеяны по всему миру, каждый объект-прибор содержит объект-местоположение, который дает информацию о его расположении и часовом поясе. Вы хотите, чтобы ваши пользователи могли выбирать прибор и наносить его данные на график с отметкой часового пояса. Вы можете записать

```
public void plotDate(Date aDate Selection aSelection) {
  TimeZone tz =
```

```
    ASelection.getRecorder().getLocation().getTimeZone();
...
}
```

Но теперь подпрограмма построения графика без особой надобности связана с тремя классами – Selection, Recorder и Location. Этот стиль программирования резко увеличивает число классов, от которых зависит наш класс. Почему это плохо? Потому что при этом увеличивается риск того, что внесение несвязанного изменения в другой части системы затронет вашу программу. Например, если сотрудник по имени Фред вносит изменение в класс Location так, что он непосредственно более не содержит TimeZone, то вам придется внести изменения и в свою программу.

Вместо того чтобы продираться через иерархию самостоятельно, просто спросите напрямую о том, что вам нужно:

```
public void plotDate(Date aDate, TimeZone aTz) {
  ...
}
plotDate(someDate, someSelection.getTimeZone());
```

Мы добавили метод к классу Selection, чтобы получить часовой пояс от своего имени; подпрограмме построения графика неважно, передается ли часовой пояс непосредственно из класса Recorder, от некоего объекта, содержащегося в Recorder, или же класс Selection сам составляет другой часовой пояс. В свою очередь, подпрограмма выбора должна запросить прибор о его часовом поясе, оставив прибору право получить его значение из содержащегося в нем объекта Location.

Непосредственное пересечение отношений между объектами может быстро привести к комбинаторному взрыву [28] отношений зависимости. Признаки этого явления можно наблюдать в ряде случаев:

1. В крупномасштабных проектах на языках C или C++, где команда компоновки процедуры тестирования длиннее, чем сама программа тестирования.

2. «Простые» изменения в одном модуле, распространяющиеся в системе через модули, не имеющие связей.

3. Разработчики, которые боятся изменить программу, поскольку они не уверены, как и на чем скажется это изменение.

Системы, в которых имеется большое число ненужных зависимостей, отличаются большой сложностью (и высокими затратами) при сопровождении и в большинстве случае весьма нестабильны. Для того чтобы поддерживать число зависимостей на минимальном уровне, мы воспользуемся законом Деметера при проектировании методов и функций.

Закон Деметера для функций

Закон Деметера для функций [LH89] пытается свести к минимуму связывание между модулями в любой программе. Он пытается удержать вас от проникновения в объект для получения доступа к методам третьего объекта. Краткое содержание данного закона представлено на рисунке 5.1.

Создавая «скромную» программу, в которой закон Деметера соблюдается в максимально возможной степени, мы можем добиться цели, выраженной в следующей подсказке:

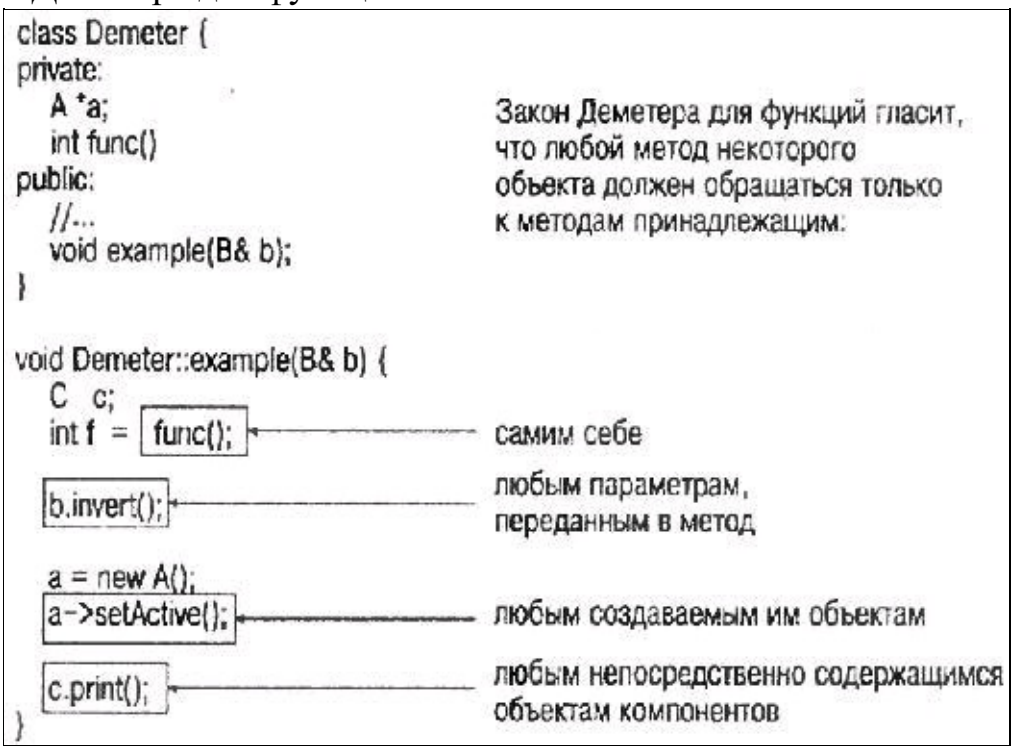
Подсказка 36: Минимизируйте связывание между модулями

А не все ли равно?

Оказывает ли следование закону Деметера (каким бы хорошим он не был с точки зрения теории) реальную помощь в создании программ, более простых в сопровождении?

Исследования [BBM96] показали, что классы в языке C++ с большими совокупностями откликов менее ошибкоустойчивы, чем классы с небольшими совокупностями (совокупность откликов представляет собой число функций, непосредственно вызываемых методами конкретного класса).

Рис. 5.1. Закон Деметера для функций



Поскольку следование закону Деметера уменьшает размер совокупности отклика в вызывающем отклике, то классы, спроектированные данным образом, также будут менее склонны к наличию ошибок (см. [URL 56], где приводится более подробная информация о статьях и других источниках по проекту Деметера).

Использование закона Деметера сделает вашу программу более адаптируемой и устойчивой, но не бесплатно: будучи "генеральным подрядчиком", ваша программа должна непосредственно делегировать полномочия и управлять всеми существующими субподрядчиками, не привлекая к этому клиентов вашего модуля. На практике это означает, что вы будете создавать большое количество методов-оболочек, которые просто направляют запрос далее к делегату. Эти методы-оболочки влекут за собой расходы во время исполнения и накладные расходы дискового пространства, которые могут оказаться весьма значительными, а для некоторых приложений даже запредельными.

Как и при использовании любой методики, вы должны взвесить все «за» и «против» для конкретного приложения. В проекте схемы базы данных обычной практикой является «денормализация» схемы для улучшения производительности: нарушение правил нормализации в обмен на скорость выполнения. Подобного же компромисса можно достичь и в этом случае. На самом деле, обращая закон Деметера и плотно связывая несколько модулей, вы можете получить существенный выигрыш в производительности. Ваша конструкция работает прекрасно, пока она известна и приемлема для этих связываемых модулей.

Физическая несвязанность

В данном разделе мы много говорим о сохранении логической несвязанности между элементами проектируемой системы. Однако существует взаимозависимость другого рода, которая становится весьма существенной с увеличением масштаба систем. В своей книге "Large-Scale C++ Software Design" [Lak96] Джон Лакос обращается к вопросам, касающимся отношений между файлами, каталогами и библиотеками, составляющими систему. Игнорирование этих проблем физического проектирования в крупномасштабных проектах приводит, помимо прочих проблем, к тому, что цикл сборки может растягиваться на несколько дней, а процедуры модульного тестирования могут сорвать сроки готовности всей системы. Г-н Лакос приводит убедительные доказательства того, что логическое и физическое проектирование должно осуществляться в тандеме и что устранение повреждений в большом фрагменте программы, нанесенных ему циклическими зависимостями, представляется чрезвычайно трудным делом. Мы рекомендуем вам прочесть эту книгу, если вы участвуете в разработке крупномасштабных проектов, даже если вы осуществляете реализацию на языке, отличном от C++.

В противном случае вы можете оказаться на пути к хрупкому, негибкому будущему. Или вообще оказаться без будущего.

Другие разделы, относящиеся к данной теме:

- Ортогональность
- Обратимость
- Проектирование по контракту
- Балансировка ресурсов
- Всего лишь визуальное представление
- Команды прагматиков
- Безжалостное тестирование

Вопросы для обсуждения

• Мы обсудили, как делегирование полномочий облегчает соблюдение закона Деметера и, следовательно, уменьшает связывание. Однако написание всех методов, необходимых для пересылки вызовов к делегированным классам, является утомительной процедурой, чреватой ошибками. Каковы преимущества и недостатки написания препроцессора, который автоматически генерирует эти вызовы? Должен ли этот препроцессор запускаться только единожды, или же он должен применяться как составная часть процесса сборки?

Упражнения

24. Мы обсудили концепцию физической несвязанности в последней врезке. Какой из указанных ниже файлов заголовка в языке C++ характеризуется более сильным связыванием с остальной системой? (Ответ см. в Приложении B.)

person1.h

```
#include "date.b"
class Person 1 {
private:
  Date myBirthdate;
public:
  Person1(Date &birthDate);
//...
```

person2.h

```
class Date;
class Person2 {
private:
   Date *myBirthdate;
public:
```

25. В данном примере и примерах из упражнений 26 и 27 определите, являются ли показанные вызовы метода допустимыми с точки зрения закона Деметера. Первый пример написан на языке Java. (Ответом, в Приложении В.)

```
public void showBalance(BankAccount acct) {
  Money amt = acct.getBalance();
  printToScreen(amt.printFormat());
}
```

26. Этот пример также написан на языке Java. (Ответ см. в Приложении В.)

```
public class Colada {
  private Blender myBlender;
  private Vector myStuff;
  public Colada() {
    myBlender = new Blender();
    myStuff = new Vector));
  }
  private void doSomething() {
     myBlender.addlngredients(myStuff.elements());
  }
}
```

27. Этот пример написан на языке C + +. (Ответ см. в Приложении В.)

```
void processTransaction(BankAccount acct, int) {
  Person *who;
  Money amt;
  amt.setValue(123.45);
  acct.setBalance(amt);
  who = acct.getOwnerQ;
  markWorkflow(who->name(), SET BALANCE);
}
```

27
Метапрограммирование

Никакая гениальность не спасает от любви к подробностям.

Восьмой закон Леви

Подробности смешивают все в нашей первоначальной программе – особенно если эти подробности часто меняются. Каждый раз, когда нам приходится входить в программу и вносить в нее изменения для того, чтобы привести ее в соответствие с изменившейся бизнес-логикой, законодательством или вкусами руководства, мы рискуем нарушить систему, т. е. внести в нее новый дефект.

Поэтому мы говорим: "Долой подробности!". Уберите их из программы. В этом случае мы можем сделать нашу программу гибкой при настройке и легко адаптирующейся к изменениям.

Динамическая конфигурация

Прежде всего мы хотим сделать системы гибкими при настройке. Это касается не только цвета экрана и текста, но и более глубоких вещей, таких как выбор алгоритмов, программ баз данных, технологии связующего программного обеспечения и стиля пользовательского интерфейса. Эти пункты должны реализовываться в виде вариантов конфигурации, а не за счет интеграции или технологии.

Подсказка 37: Осуществляйте настройку, а не интеграцию

Используйте метаданные для спецификации вариантов настройки приложения: подгонки параметров, глобальных параметров пользователя, каталога, в который производится установка приложения, и т. д.

Так что же такое метаданные? Строго говоря, метаданные – это данные о данных. Наиболее распространенным примером, вероятно, является схема базы данных или словарь данных. Схема содержит данные, которые описывают поля (столбцы) в терминах имен, длины и других атрибутов. Вы должны иметь возможность доступа к этой информации и ее обработки так, как если бы это были любые другие данные в этой базе.

Мы используем этот термин в самом широком смысле. Метаданные – это любые данные, которые описывают приложение – как оно выполняется, какие ресурсы обязано использовать и т. д. Обычно доступ к данным и их использование осуществляется на этапе выполнения, а не компиляции. Вы используете метаданные все время, по крайней мере, это делают ваши программы. Предположим, вы щелкаете мышью для того, чтобы скрыть панель инструментов в интернет-браузере. Браузер будет сохранять эти глобальные параметры как метаданные в своего рода внутренней базе данных.

Эта база данных может быть сформирована в собственном формате или может воспользоваться стандартным механизмом. При работе в операционной системе Windows таким механизмом является либо файл инициализации (используется суффикс .ini), либо записи в системном реестре. При работе с Unix подобная функциональная возможность обеспечивается

системой X Window с помощью файлов Application Default. Java использует файлы Property. Во всех этих средах для извлечения значения вы указываете ключ. В других, более мощных и гибких реализациях метаданных используется встроенный язык сценариев (см. "Языки, отражающие специфику предметной области").

При реализации этих глобальных параметров в браузере Netscape фактически использованы обе эти технологии. В версии 3 параметры сохранялись в виде пар "ключ-значение":

```
SHOWTOOLBAR: False
```

В версии 4 параметры больше напоминали JavaScript:

```
user_pref("custtoolbar.Browser.Navigation_Toolbar.open", false);
```

Приложения, управляемые метаданными

Но мы хотим большего, нежели использовать метаданные для простых глобальных параметров. Мы хотим настраивать и управлять приложением через метаданные – насколько это возможно. Наша цель – думать описательно (обозначая, что должно быть сделано, а не как это должно быть сделано) и создавать высокодинамичные и адаптируемые программы. Это можно сделать, придерживаясь общего правила: программировать для общего случая и помещать всю специфику в другое место – за пределы компилируемого ядра программы.

> Подсказка 38: Помещайте абстракции в текст программы, а подробности – в область метаданных

Этот подход характеризуется несколькими преимуществами:

• Он вынуждает вас делать конструкцию несвязанной, что приводит к созданию более гибкой и адаптируемой программы.

• Он заставляет вас создавать более устойчивую, абстрактную конструкцию за счет отнесения подробностей, выводя все подробности за пределы программы.

• Вы можете настроить приложение, не прибегая к его перекомпиляции. Вы также можете использовать этот уровень настройки для обеспечения обходных путей при критических дефектах систем, находящихся в эксплуатации.

• Метаданные могут быть выражены способом, который находится намного ближе к предметной области, по сравнению с универсальным языком программирования (см. "Языки, отражающие специфику конкретной области").

• Вы даже сможете реализовывать несколько различных проектов, используя то же самое ядро приложения, но с различными метаданными.

Как правило, нам хочется отложить определение большинства подробностей на последний момент и оставить их как можно менее сложными для изменения. Создавая решение, позволяющее нам вносить изменения быстро, мы можем лучше справляться с потоком направленных сдвигов, которые погубили многие проекты (см. "Обратимость").

Бизнес-логика

Итак, мы выбрали механизм базы данных в качестве опции настройки и предусмотрели метаданные для определения стиля пользовательского интерфейса. Можем ли мы сделать большее? Несомненно.

Поскольку стратегия и бизнес-правила подвергнутся изменениям скорее, нежели любые другие аспекты проекта, есть смысл поддерживать их в очень гибком формате.

Например, приложение, автоматизирующее процесс закупок, может включать в себя различные корпоративные стратегии. Может быть, вы производите оплату небольшим фирмам-поставщикам через 45 дней, а большим – через 90 дней. Сделайте настраиваемыми определения типов поставщиков, а также самих периодов времени. Используйте возможность обобщения.

Возможно, вы создаете систему с ужасающими требованиями к последовательности операций. Действия начинаются и заканчиваются согласно сложным (и изменяющимся) бизнес-правилам. Подумайте об их реализации в виде некой системы на основе правил (или экспертной системы), встроенных в ваше приложение. Тем самым вы осуществите его настройку за счет написания правил, а не программы.

Менее сложная логика может быть выражена при помощи мини-языка, что делает необязательным повторную компиляцию и развертывание при изменении среды. Пример приведен в разделе "Языки, отражающие специфику предметной области".

Когда осуществлять настройку

Как было упомянуто в разделе "Преимущество простого текста", рекомендуется представлять метаданные о настройке в формате простого текста – это делает жизнь проще.

Но когда программа должна осуществлять считывание этой настройки? Многие программы осуществляют просмотр только при неудачном запуске. Если вам необходимо изменить настройку, это вынуждает вас перезапускать приложение. Более гибким подходом является написание программ, которые могут перезагружать свои настройки во время выполнения. Но эта гибкость обходится недешево: она более сложна в реализации.

Рассмотрим, как будет использоваться приложение: если это продолжительный серверный процесс, то вам понадобится некий механизм для повторного считывания и применения метаданных в ходе выполнения программы. Для небольшого клиентского приложения с графическим интерфейсом, которое перезапускается достаточно быстро, это может и не понадобиться.

Данное явление не ограничивается прикладными программами. Все мы раздражаемся, если операционные системы заставляют нас проводить перезагрузку при установке простых приложений или изменении совершенно безвредного параметра.

Пример: пакет Enterprise Java Beans

Пакет EJB (Enterprise Java Beans) является интегрированной средой, предназначенной для упрощения программирования в распределенной среде, основанной на транзакциях. Этот пакет упоминается в связи с тем, что он иллюстрирует использование метаданных для настройки приложений и упрощения процедуры написания программы.

Предположим, что вы хотите создать некоторую программу на языке Java, которая будет принимать участие в транзакциях на различных машинах, с базами данных от различных производителей и с разными моделями потоков и распределения нагрузки.

Хорошая новость: вам не нужно беспокоиться обо всем этом. Вы пишете так называемый bean-элемент – отдельный объект, который следует определенным соглашениям, и помещаете его в контейнер bean-элементов, управляющий многими низкоуровневыми средствами от вашего имени. Вы можете писать программу для bean-элемента, не включая какие-либо транзакционные операции или управление потоками; пакет EJB использует метаданные для указания способа обработки транзакций.

Назначение потока и распределение нагрузки указываются как метаданные для основной службы транзакций, используемой контейнером. Это разделение допускает большую гибкость при динамической настройке среды во время работы.

Контейнер bean-элемента может управлять транзакциями от имени bean-элемента одним из нескольких различных способов (включая вариант управления собственными обновлениями и отменой транзакций). Все параметры, воздействующие на поведение bean-элемента, указаны в описателе развертывания последнего – объекте, преобразованном в последовательную форму и содержащем нужные метаданные.

Распределенные системы, подобные EJB, прокладывают путь в новый мир – мир настраиваемых, динамичных систем.

Совместная настройка

Выше уже говорилось о пользователях и разработчиках, настраивающих динамические приложения. Но что происходит, если вы позволяете приложениям настраивать друг друга? Речь идет о программах, которые адаптируются к операционной среде. Незапланированная, импровизированная настройка существующего программного обеспечения является мощной концепцией.

Операционные системы уже способны подстраивать себя при загрузке под аппаратное обеспечение, а web-браузеры автоматически обновляются, инсталлируя новые компоненты.

Большие приложения, с которыми вы работаете, имеют проблемы с управлением различными версиями данных и различными версиями библиотек и операционных систем. Возможно, здесь будет полезен более динамичный подход.

Не пишите нежизнеспособных программ

В отсутствие метаданных ваша программа не является столь адаптируемой или гибкой, какой она могла бы стать в противном случае. Плохо ли это? В реальном мире виды, которые не могут адаптироваться, умирают.

Птицы додо не смогли приспособиться к присутствию людей и домашних животных на острове Маврикий и быстро вымерли [29]. Это было первое документально подтвержденное исчезновение вида от рук человека.

Не дайте вашему проекту (или карьере) повторить судьбу птицы додо.

Другие разделы, относящиеся к данной теме:

- Ортогональность
- Обратимость
- Языки, отражающие специфику предметной области
- Преимущества простого текста

Вопросы для обсуждения

• Работая над текущим проектом, подумайте о следующем: какая часть программы может быть убрана из нее и перемещена в область метаданных. Как в итоге будет выглядеть «ядро» программы? Сможете ли вы повторно использовать это ядро в контексте иного приложения?

Упражнения

28. Что из нижеследующего лучше представить в виде фрагмента программы, а что вывести за ее пределы в область метаданных?

1. Назначения коммуникационных портов
2. Поддержка выделения синтаксиса различных языков в программе редактирования
3. Поддержка редактора для различных графических устройств
4. Конечный автомат для программы синтаксического анализа или сканера
5. Типовые значения и результаты, используемые в тестировании модулей

28
Временное связывание

Временное связывание – о чем это? – спросите вы. Это – о времени.

Время – аспект, который часто игнорируется в архитектуре программного обеспечения. Единственный временной параметр, который занимает наш ум – это время выполнения проекта, время, оставшееся до отправки продукта заказчику, но здесь разговор не об этом, а о роли временного фактора как элемента проектирования самого программного обеспечения. Существует два временных аспекта, представляющих для нас важность: параллелизм (события, происходящие в одно и то же время) и упорядочивание (относительное положение событий во времени).

Обычно мы не приступаем к программированию, держа в голове тот или иной аспект. Когда люди садятся за проектирование, разработку архитектуры или написание программы, события стремятся к линейности. Это и есть способ мышления большинства людей – сначала сделать «это», а потом всегда сделать «то». Но этот способ мышления приводит к связыванию во времени. Метод A всегда вызывается перед методом B; одновременно должен формироваться только один отчет; необходимо подождать перерисовки экрана до получения отклика на щелчок мыши. «Тик» обязан происходить раньше, чем "так".

Этот подход не отличается большой гибкостью и реализмом.

Нам приходится учитывать параллелизм [30] и думать о несвязанности любых временных или упорядоченных зависимостей. При этом мы выигрываем в гибкости и уменьшаем любые зависимости, основанные на времени во многих областях разработки: анализе последовательности операций, архитектуре, проектировании и развертывании.

Последовательность операций

При работе над многими проектами, нам приходится моделировать и анализировать последовательности операций пользователей, что является частью анализа требований. Мы хотели бы выяснить, что может происходить одновременно, а что – в строгой последовательности. Одним из способов осуществить задуманное является создание диаграммы последовательностей, с помощью системы обозначений наподобие языка UML (унифицированного языка моделирования) [31].

Диаграмма состоит из совокупности действий, изображенных в виде прямоугольников с закругленными уголками. Стрелка, выходящая из одной операции, идет либо к другой операции (которая может начаться после того, как первая закончится) либо к жирной линии, называемой полосой синхронизации. Как только все операции, направленные к полосе синхронизации, завершаются, можно перемещаться по стрелкам, идущим от полосы синхронизации. Операция, на которую не указывают никакие стрелки, может быть начата в любой момент.

Вы можете использовать диаграммы, чтобы добиться максимального параллелизма, определив те процессы, которые могли бы осуществляться параллельно, но не осуществляются.

Подсказка 39: Анализируйте последовательность операций для увеличения параллелизма

Например, в проекте блендера для коктейлей (упражнение 17) пользователи могут вначале описать последовательность операций следующим образом:

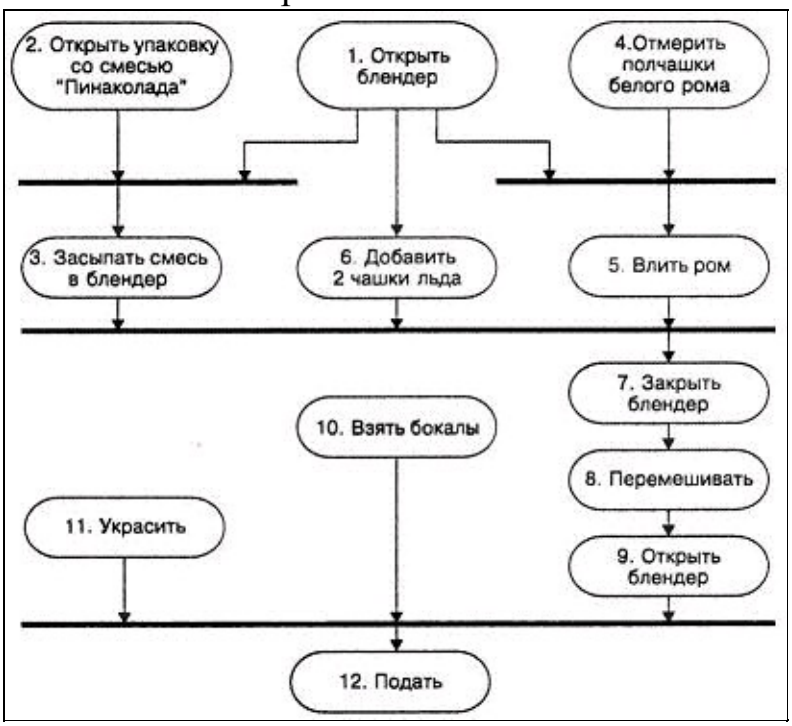
1. Открыть блендер
2. Открыть упаковку со смесью "Пинаколада"
3. Засыпать смесь в блендер
4. Отмерить полчашки белого рома
5. Влить ром
6. Добавить 2 чашки льда
7. Закрыть блендер
8. Перемешивать в течение 2 мин
9. Открыть блендер
10. Взять бокалы
11. Украсить
12. Налить

Хотя они описывают эти операции последовательно (и даже могут выполнять их последовательно), заметим, что многие из них могли бы выполняться параллельно, как показано на блок-схеме (см. рис. 5.2).

Это может открыть вам глаза на реально существующие зависимости. В этом случае задачи высшего уровня приоритета (1, 2, 4, 10 и 11) могут выполняться параллельно, как бы авансом. Задачи 3, 5 и 6 могут выполняться параллельно, но позже.

Если бы вы участвовали в конкурсе по приготовлению коктейлей «Пинаколада», эти оптимальные решения выгодно отличали бы вас от всех остальных.

Рис. 5.2. Диаграмма на языке UML: приготовление коктейля "Пинаколада"



Архитектура

Несколько лет назад мы написали систему оперативной обработки транзакций (OLAP – on-line transaction processing). В простейшем варианте все, что должна была сделать система, – это

принять запрос и обработать транзакцию в сравнении с БД. Но мы написали трехзвенное, многопроцессорное распределенное приложение: каждый компонент представлял собой независимую единицу, которая выполнялась параллельно со всеми другими компонентами. Хотя при этом возникает впечатление большой работы, это не так: при написании этого приложения мы использовали преимущество временной несвязанности. Рассмотрим этот проект более подробно.

Система принимает запросы от большого числа каналов передачи данных и обрабатывает транзакции в рамках БД.

Проект налагает следующие ограничения:

• Операции с БД занимают сравнительно большое время.

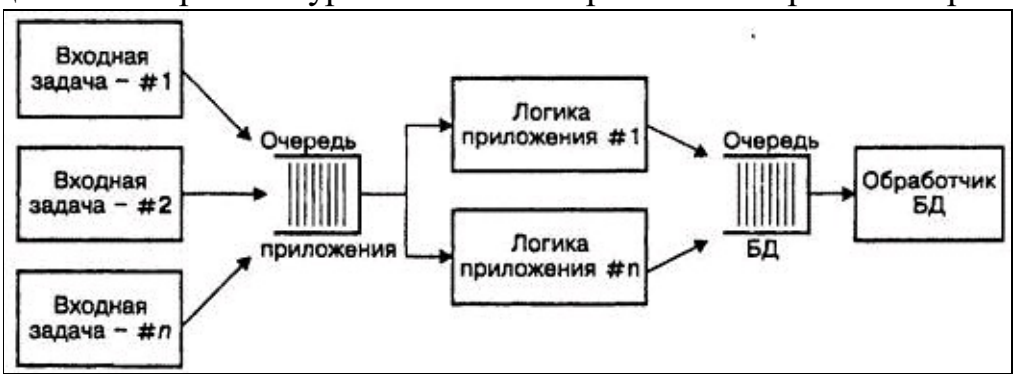
• При каждой транзакции мы не должны блокировать коммуникационные службы в момент обработки транзакции БД.

• Производительность базы ухудшается за счет слишком большого числа параллельных сеансов.

• Множественные транзакции осуществляются параллельно на каждой линии передачи данных.

Решение, обеспечивающее наилучшую производительность и самый четкий интерфейс, выглядит подобно представленному на рисунке 5.3.

РИС. 5.3. Общая схема архитектуры системы оперативной обработки транзакций



Каждый прямоугольник обозначает отдельный процесс; процессы связываются через очереди работ. Каждый входной процесс отслеживает состояние одного входного канала связи и осуществляет запросы к серверу приложения. Все запросы являются асинхронными: как только входной процесс осуществляет текущий запрос, он сразу же возвращается к отслеживанию канала на наличие трафика. Точно так же сервер приложения осуществляет запросы процесса БД [32] и уведомляется в момент завершения отдельной транзакции.

На этом примере также демонстрируется способ быстрого и грубого распределения нагрузки между множественными потребительскими процессами: это так называемая модель голодного потребителя.

В модели голодного потребителя центральный планировщик заменяется на несколько независимых задач потребителя и централизованную очередь работ. Каждая задача потребителя захватывает некий фрагмент очереди работ и продолжает заниматься своим делом – его обработкой. Как только задача заканчивает свою работу, она возвращается к очереди за новой порцией. В этом случае, если выполнение какой-либо задачи срывается, другие задачи могут "натянуть поводья" и каждый отдельный компонент может продолжаться в своем собственном темпе. Происходит временная несвязанность одного компонента с другими.

На самом деле, вместо компонентов мы создали службы – независимые, параллельные объекты, скрытые за четко определенными, непротиворечивыми интерфейсами.

Проектирование с использованием принципа параллелизма

Поскольку Java все чаще принимается в качестве платформы, многие разработчики перешли к многопоточному программированию. Но программирование с использованием потоков налагает на конструкцию некоторые ограничения – и это хорошо. Эти ограничения настолько полезны, что нам хотелось бы пребывать под их благодатным покровом, когда бы мы ни занимались написанием программ. Это поможет нам делать нашу программу несвязанной и бороться с так называемым "программированием в расчете на стечение обстоятельств" (см. ниже одноименный раздел).

При работе с линейной программой легко сделать предположения, которые в конечном итоге приведут к небрежно написанным программам. Но параллелизм заставляет задумываться о происходящем несколько глубже – вы больше не находитесь в безвоздушном пространстве. Поскольку многие события могут теперь происходить "в одно и то же время", вы можете внезапно столкнуться с зависимостями, основанными на факторе времени. Прежде всего необходимо защитить любые глобальные или статические переменные от параллельного доступа. Теперь можно задать самому себе вопрос, зачем нужна глобальная переменная на первом месте. Кроме того, необходимо убедиться в том, что вы предоставляете непротиворечивую информацию о состоянии независимо от порядка вызовов. Например, в какой момент допускается опрашивание состояния вашего объекта? Если ваш объект находится в недопустимом состоянии в период между определенными вызовами, то вы, вероятно, полагаетесь на стечение обстоятельств – никто не вызовет ваш объект в этот момент времени.

Предположим, что есть подсистема работы с окнами, в которой интерфейсные элементы вначале создаются, а затем отображаются на дисплее. Вам не разрешается задавать состояние в элементе, пока он не отобразится. В зависимости от заданных параметров программы вы можете полагаться на то условие, что ни один другой объект не может воспользоваться созданным элементом, пока вы не выведете его на дисплей.

Но в параллельной системе это может и не выполняться. При вызове объекты всегда обязаны находиться в допустимом состоянии, а они могут вызываться в самое неподходящее время. Вы обязаны убедиться, что объект находится в допустимом состоянии в любой момент, когда потенциально он может быть вызван. Зачастую эта проблема возникает с классами, которые определяют отдельные программы конструктора и инициализации (где конструктор не оставляет объект в инициализированном состоянии). Используя инварианты класса, обсуждаемые в разделе "Проектирование по контракту", вы сможете избежать этой ловушки.

Четкие интерфейсы

Размышления о параллелизме и зависимостях, упорядоченных во времени, могут заставить вас проектировать более четкие интерфейсы. Рассмотрим библиотечную подпрограмму на языке C под названием strtok, которая расщепляет строку на лексемы.

Конструкция strtok не является поточно-ориентированной [33], но это не самое плохое,

рассмотрим временную зависимость. Первый раз вы обязаны вызвать подпрограмму Strtok с переменной, которую вы хотите проанализировать, а во всех последующих вызовах использовать NULL вместо этой переменной. Если переменная принимает значение, отличное от NULL, программа повторно производит разбор содержимого буфера. Не принимая во внимание потоки, предположим, что вы собираетесь использовать Strtok для одновременного синтаксического анализа двух отдельных строк:

```
char buf1[BUFSIZ];
char buf2[BUFSIZ];
char *p, *q;
strcpy(bufl, "это тестовая программа");
strcpy(buf2, "которая не будет работать");
p = strtck(buf1," ");
q = strtok(buf2," ");
while (p && q) {
  printf("%s %s\n", p, q);
  p = strtok(NULL, " ");
  q = strtok(NULL, " ");
}
```

Представленная программа работать не будет: существует неявное состояние, сохраняющееся в strtok между запросами. Вам придется использовать Strtok одновременно только с одним буфером.

Конструкция синтаксического анализатора строк на языке Java будет отличаться от указанной выше. Она должна быть поточно-ориентированной и представлять непротиворечивое состояние.

```
StringTokenizer st1 = new StringTokenizer("this is a test");
StrJngTokenlzer st2 = new StringTokenizer("this test will work");
while (st1.hasMoreTokens() && st2.hasMoreTokens()) {
  System.out.println(st1.nextToken());
  System.out.println(st2.nextToken());
}
```

Программа StringTokenizer обладает более четким и простым в сопровождении интерфейсом. Она не содержит в себе никаких сюрпризов и в будущем не приводит к появлению таинственных дефектов, чего нельзя сказать о программе Strtok.

Подсказка 41: При проектировании всегда есть место параллелизму

Развертывание

Как только вы спроектировали архитектуру с элементом параллельности, задача об управлении многими параллельными службами упрощается: модель становится всеобъемлющей.

Теперь вы можете проявить гибкость относительно способа развертывания приложения: по автономной модели, модели «клиент-сервер» или по n-звенной модели. Создавая архитектуру системы на основе независимых служб, вы также придаете динамизм процессу конфигурирования. Рассчитывая на параллелизм и разделяя операции во времени, вы получаете

вес эти варианты, включая автономный вариант развертывания, где вы можете отказаться от параллелизма.

Другой путь (попытка внести параллелизм в непараллельное приложение) представляется намного сложнее. Если мы проектируем с учетом параллелизма, то со временем нам легче обеспечивать расширяемость и производительность, а если этот момент не настает, мы все равно получаем выгоду от более четкого интерфейса.

Так, может быть, пора?

Другие разделы, относящиеся к данной теме:

- Проектирование по контракту
- Программирование в расчете на стечение обстоятельств

Вопросы для обсуждения

- Сколько задач вы выполняете параллельно, готовясь к работе? Можете ли вы выразить это с помощью диаграммы на языке UML? Можете ли вы найти иной, более быстрый способ подготовки к работе, придав своим действиям больший параллелизм?

29
Всего лишь визуальное представление

Каждый смертный все же видит
Только то, что хочет видеть,
Отметая остальное.
Ля-ля-ля…

П. Саймон и А. Гарфункель, Боксер

Ранее нас учили не писать программы одним большим куском, а использовать принцип "разделяй и властвуй" и разбивать программу на модули. У каждого молу-ля есть свои собственные обязанности; модуль (или класс) считается четко определенным, если у него имеется одна четко обозначенная обязанность.

Но как только вы разбиваете программу на различные модули, основанные на обязанностях, вы сталкиваетесь с новой проблемой. Каким образом объекты общаются друг с другом на стадии выполнения программы? Как вы управляете логическими зависимостями между ними? Другими словами, как вы осуществляете синхронизацию изменений состояния (или обновление значений данных) различных объектов? Этой работе должна быть присуща четкость и гибкость – мы не хотим, чтобы они узнали друг о друге слишком много. Мы хотим, чтобы каждый модуль был похож на героя песни Саймона и Гарфункеля и видел только то, что хочет увидеть.

Начнем с концепции события. Событие представляет собой специальное сообщение, в котором говорится: "Только что случилось нечто интересное" (разумеется, с точки зрения наблюдателя). Мы можем использовать события, чтобы сигнализировать одному объекту об изменениях, произошедших с другим объектом, в которых последний может быть заинтересован.

Подобное использование событий сводит к минимуму связывание между двумя объектами – отправителю события не нужно обладать явной информацией о получателе. На самом деле могут существовать и множественные получатели, каждый из которых сосредоточен на собственном перечне основных операций (отправитель же находится в блаженном неведении относительно этого факта).

Однако при использовании событий необходимо соблюдать некоторую осторожность. Например, в одной из ранних версий Java одна подпрограмма получила все события, предназначенные для специфического приложения. Это не совсем подходит для облегчения сопровождения или развития программы.

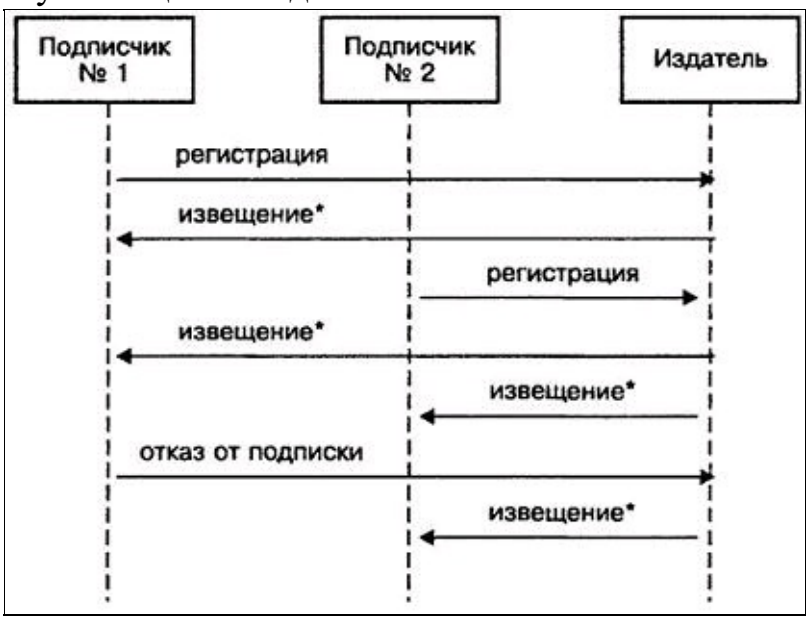
Протокол "Публикация и подписка"

Почему считается дурным тоном пропускать все события через одну-единственную программу? Потому что при этом нарушается инкапсулирование объекта – теперь этой подпрограмме приходится получать сокровенную информацию о взаимодействии между многими объектами. Это также способствует увеличению связывания, а мы пытаемся его уменьшить. Поскольку и самим объектам приходится получать информацию об этих событиях, то, по всей вероятности, вы собираетесь нарушить принцип DRY, принцип ортогональности и, может быть, некоторые разделы Женевской конвенции. Быть может, вам случалось видеть подобные программы – их доминантой является огромный оператор case или многообразная конструкция if-then. Мы можем сделать это изящнее.

Объекты должны иметь возможность регистрации только для приема событий, которые им нужны, и никогда не должны посылать события, которые им не нужны. Мы не хотим, чтобы наши объекты подверглись спаммингу! Вместо этого мы можем воспользоваться протоколом типа "публикация и подписка", который представлен на рисунке 5.4 с помощью диаграммы последовательностей на языке UML [34].

На блок-схеме последовательности показан поток сообщений между несколькими объектами, которые располагаются по столбцам. Каждое сообщение обозначено стрелкой с текстом, идущей от столбца отправителя к столбцу получателя. Звездочка у текста означает, что возможна посылка более одного сообщения данного типа.

Рис. 5.4. Протокол "Публикация и подписка"



Если нам интересны определенные события, которые генерируются объектом Publisher (Издатель), то все, что нам нужно, – это зарегистрироваться. Объект Publisher отслеживает все заинтересованные объекты Subscriber (Подписчик); когда объект Publisher генерирует событие, представляющее интерес, он, в свою очередь обращается к каждому объекту Subscriber, извещая их о том, что данное событие произошло.

На эту тему существует несколько вариаций, отражающих другие стили обмена данными. Объекты могут использовать протокол "Публикация и подписка" на одноранговой основе (как показано выше), а также "программную шину", где централизованный объект поддерживает базу данных «слушателей» и осуществляет соответствующую диспетчеризацию. Вы даже можете получить схему, в которой критические события транслируются ко всем «слушателям» – как зарегистрированным, так и незарегистрированным. Одна из возможных реализаций событий в распределенной среде иллюстрируется службой сообщений CORBA, описанной во врезке "Служба событий CORBA" (см. ниже).

Можно использовать протокол "Публикация и подписка" для реализации очень важного принципа проектирования: отделения самой модели от ее визуальных представлений. Начнем с примера графического интерфейса, используя конструкцию на языке Smalltalk, где зародилась данная концепция.

Принцип "модель-визуальное представление-контроллер»

Предположим, что есть приложение – электронная таблица. В дополнение к числам,

расположенным в самой таблице, также имеется график, отображающий числа на гистограмме и диалоговое окно суммы с накоплением, отображающим сумму чисел в некотором столбце таблицы.

Служба событий CORBA

Служба событий CORBA позволяет объектам-участникам отправлять и получать уведомления о событиях через общую шину, так называемый канал событий. Канал событий принимает решение по обработке событий, а также осуществляет разделение производителей и потребителей событий. Он работает в двух основных режимах: «проталкивание» и "вытягивание".

В режиме «проталкивания» поставщики событий информируют канал событий о том, что событие произошло. Затем канал автоматически распространяет это событие ко всем объектам-клиентам, которые зарегистрировались, выражая свой интерес.

В режиме «вытягивания» клиенты периодически опрашивают канал событий, который в свою очередь, опрашивает поставщика, предлагающего данные о событии в соответствии с запросом.

Хотя служба событий CORBA может использоваться для реализации всех событийных моделей, описанных в данном разделе, ее можно рассматривать и в другом качестве. CORBA облегчает связь между объектами, написанными на различных языках программирования и выполняющимися на географически рассредоточенных машинах с различными архитектурами. Находясь на верхнем уровне CORBA, служба событий предоставляет вам способ, отличающийся отсутствием связанности и позволяющий взаимодействовать с приложениями, разбросанными по всему миру и написанными людьми, которых вы никогда не встречали, и пишущими на языках, о которых вы и знать не знаете.

Очевидно, мы не хотим иметь три отдельных копии одних и тех же данных. Поэтому мы создаем модель – сами данные и обычные операции для их обработки. Затем мы можем создать отдельные визуальные представления, которые отображают данные различными способами: в виде электронной таблицы, графика или поля суммы с накоплением. Каждое из этих визуальных представлений может располагать собственными контроллерами. Например, график может располагать неким контроллером, позволяющим приближать и отдалять объекты, осуществлять панорамирование относительно данных. Ни одно из этих средств не оказывает влияния на данные, только на это представление.

Это и является ключевым принципом, на котором основана парадигма "модель-визуальное представление-контроллер": отделение модели от графического интерфейса, ее представляющего, и средств управления визуальным представлением [35].

Действуя подобным образом, вы можете извлечь пользу из некоторых интересных возможностей. Вы можете поддерживать множественные визуальные представления для одной и той же модели данных. Вы можете использовать обычные средства просмотра со многими различными моделями данных. Вы даже можете поддерживать множественные контроллеры для обеспечения нетрадиционных механизмов ввода данных.

Подсказка 42: Отделяйте визуальные представления от моделей

Ослабляя связанность между моделью и ее визуальным представлением/контроллером, вы приобретаете большую гибкость практически за бесценок. На самом деле, эта методика является одним из важнейших способов сохранения обратимости (см. "Обратимость").

Java: древовидное визуальное представление

Хорошим примером принципа "модель-визуальное представление-контроллер" является графический элемент в древовидной схеме Java. Элемент, отображающий дерево, активизируемое щелчком мыши, в действительности представляет собой набор нескольких различных классов, организованных по шаблону "модель-визуальное представление-контроллер".

Все, что вам нужно сделать для получения полнофункционального элемента дерева, – это обеспечить источник данных, который соответствует интерфейсу TreeModel. Ваша программа становится моделью дерева.

Визуальное представление создается классами TreeCellRenderer и TreeCellEditor, которые могут быть унаследованы и настроены для обеспечения различных цветов, шрифтов и пиктограмм в графическом элементе. JTree действует в качестве контроллера для элемента дерева и обеспечивает некоторую общую функциональную возможность просмотра.

Осуществив разделение модели и ее визуального представления, мы серьезно упростили процесс программирования. Уже не нужно беспокоиться об элементе дерева. Вместо этого необходимо предоставить источник данных.

Предположим, к вам подходит вице-президент фирмы и высказывает пожелание, чтобы вы быстро написали приложение, которое позволяет ему управлять структурной схемой фирмы, содержащейся в унаследованной базе данных на мэйнфрейме. Просто напишите оболочку, которая получает данные с мэйнфрейма, представляет ее в виде TreeModel, и – "Вуаля!" – у вас имеется полнофункциональный элемент дерева.

Теперь можете капризничать и начать использовать классы средств просмотра; вы можете изменять представление узлов и использовать специальные пиктограммы, шрифты или цвета. Когда вице-президент вернется к вам и скажет, что новые корпоративные стандарты требуют использования для некоторых служащих пиктограммы "Веселый Роджер", то вы можете внести изменения в TreeCellRenderer, не затрагивая другие программы.

Отходя от графических интерфейсов

Хотя принцип "модель-визуальное представление-контроллер" обычно реализуется в контексте графического интерфейса, на самом деле он является универсальной методикой программирования. Визуальное представление – это некая интерпретация модели (возможно, подмножества), и она не обязана быть графической. Контроллер в большей части является механизмом координации и не должен ассоциироваться с устройством ввода любого типа.

• **Модель.** Абстрактная модель данных, представляющая целевой объект. Модель не располагает непосредственной информацией о любых визуальных представлениях или контроллерах.

• **Визуальное представление.** Способ интерпретации модели. Оно подписывается на изменения в модели и логические события, приходящие от контроллера.

• **Контроллер.** Способ контроля визуального представления и снабжения модели новыми данными. Он осуществляет публикацию событий для модели и визуального представления.

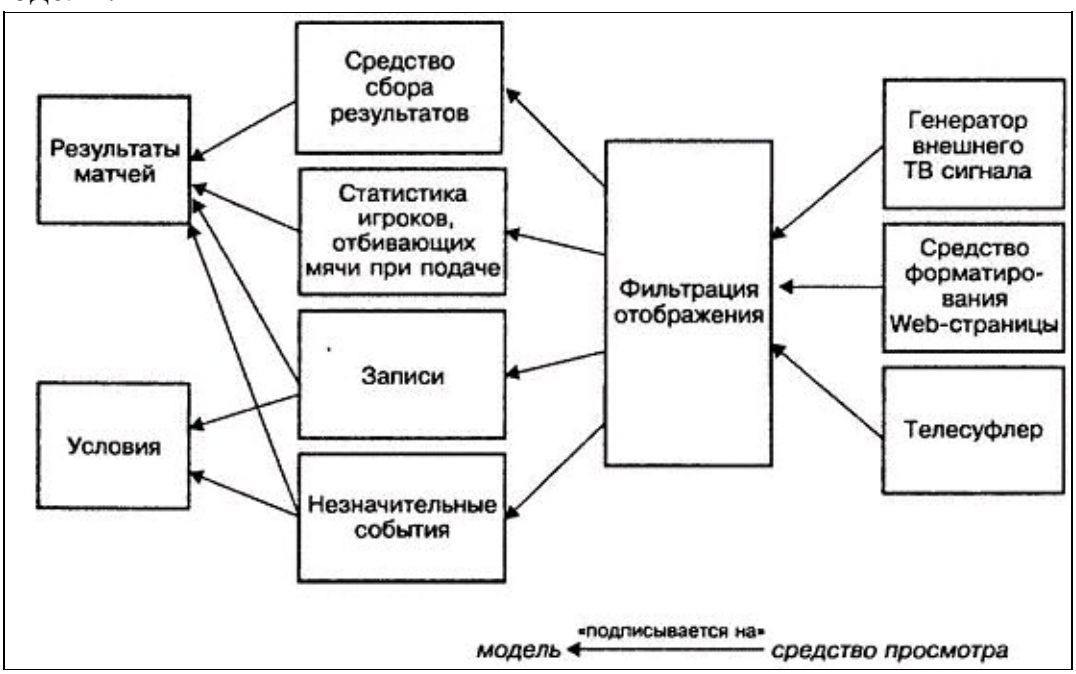
Рассмотрим пример с текстовым интерфейсом.

Игра в бейсбол представляет собой уникальное явление. Где еще можно найти такие пустяки, как "самый результативный матч, сыгранный во вторник под дождем при искусственном освещении между командами, названия которых начинаются с гласной буквы"? Предположим, что нам поручили разработать программу для помощи бесстрашным дикторам, которым по должности полагается сообщать счет, статистику и прочие мелочи.

Ясно, что нам необходима информация о матче, который проходит в настоящее время – играющие команды, условия, игрок, принимающий подачу, счет и т. д. Эти факты образуют наши модели; они будут обновляться по мере поступления новой информации (смена подающего, выбывание игрока, начался дождь…).

Затем у нас появится ряд объектов – визуальных представлений, которые будут использовать эти модели. Один объект должен наблюдать за набираемыми очками – для обновления текущего счета. Другой объект может получать уведомления о новых игроках, отбивающих мяч, и извлекать краткую справку об их статистических показателях за год. Третий объект может просматривать данные и проверять, не установлен ли мировой рекорд. Можно даже использовать средство просмотра «мелочей», которое несет ответственность за придумывание сверхъестественных и бесполезных фактов, щекочущих нервы зрителей.

Рис. 5.5. Комментирование бейсбольного матча. Средства просмотра являются подписчиками модели.



Но мы не хотим, чтобы несчастный диктор работал со всеми этими окнами непосредственно. Вместо этого мы сделаем так, чтобы каждое из окон генерировало извещения об «интересных» событиях, и обеспечим возможность планирования показа с помощью некоторого высокоуровневого объекта [36].

Эти объекты (средства просмотра) внезапно стали моделями высокоуровневого объекта, который сам по себе может стать моделью для различных форматирующих средств просмотра. Одно такое средство просмотра могло бы создать сценарий для телесуфлера, с которым работает диктор, второе могло бы генерировать заставки непосредственно на спутниковом канале, а третье могло бы осуществлять обновление web-страниц телевизионной сети или бейсбольной

команды (см. рис. 5.5).

Подобная сеть "модель-средство просмотра" является универсальной (и весьма ценной) методикой проектирования. Каждый канал связи осуществляет отделение исходных данных от событий, их породивших; каждое новое средство просмотра есть некая абстракция. И поскольку отношения представляют собой сеть (а не линейную цепь), то мы обладаем большой гибкостью. Каждая модель может включать в себя много средств просмотра, а одно средство просмотра может работать со многими моделями.

В усовершенствованных системах, наподобие описанной выше, полезно иметь окна отладки – специализированные окна, которые отображают подробности модели. Дополнение системы средством трассировки отдельных событий также способствует существенной экономии времени.

Все такой же связанный (после стольких лет)

Несмотря на то, что мы добились уменьшения связанности, прослушивающие процессы и генераторы событий (подписчики и издатели) все равно обладают некоторой информацией друг о друге. Например, в языке Java они обязаны прийти к соглашению об общих определениях интерфейса и вызовах.

В следующем разделе мы рассмотрим способы дальнейшего уменьшения степени связанности при помощи формы "публикация и подписка", в которой ни один из участников не должен знать друг о друге или обращаться напрямую друг к другу.

Другие разделы, относящиеся к данной теме:

- Ортогональность
- Обратимость
- Несвязанность и закон Деметера
- Доски объявлений
- Все эти сочинения

Упражнения

29. Предположим, что имеется система бронирования авиабилетов, основанная на следующем принципе формирования авиарейса:

```
public interface Flight {
//Return false if flight full.
public Boolean addPassenger(Passenger p);
public void addToWaitlJst(Passenger p);
public int getFlightCapacity();
public int getNumPassengers();
}
```

Если вы добавляете имя пассажира в лист ожидания авиарейса, то при появлении вакантного места ему будет предложено воспользоваться этим рейсом автоматически.

Чтобы составить расписание дополнительных рейсов, требуется большая работа с отчетами, заключающаяся в выискивании рейсов, количество мест на которых меньше или равно числу

проданных билетов. Это срабатывает, но занимает много времени.

Нам хотелось бы обладать большей гибкостью при обработке данных о пассажирах в листе ожидания и как-то решить проблемы с этим огромным отчетом – его формирование занимает слишком много времени. Воспользуйтесь идеями, изложенными в данном разделе, чтобы спроектировать этот интерфейс по-новому.

30
Доски объявлений

На стене написано…

Обычно вы не связываете понятие изящества с полицейскими детективами. Но рассмотрим пример того, как детективы используют доску объявлений для координации действий и расследования убийства.

Предположим, что главный инспектор начинает с того, что устанавливает большую доску в комнате для заседаний. На ней он пишет один-единственный вопрос:

ШАЛТАЙ-БОЛТАЙ (ПОЛ: МУЖСКОЙ, ЧЕЛОВЕК-ЯЙЦО): НЕСЧАСТНЫЙ СЛУЧАЙ ИЛИ УБИЙСТВО?

Шалтай на самом деле упал, или его толкнули? Каждый детектив может внести свою лепту в раскрытие тайны этого возможного убийства, добавляя факты, показания свидетелей, любые судебные доказательства и т. д. По мере накопления данных детектив может заметить некую связь и также поместить на доску свои наблюдения или гипотезу. Этот процесс продолжается, передается от смены к смене, в нем участвуют различные лица и агенты, пока дело не будет закрыто. Примерный вид доски представлен на рисунке 5.6.

Некоторые ключевые особенности подхода с применением доски объявлений:

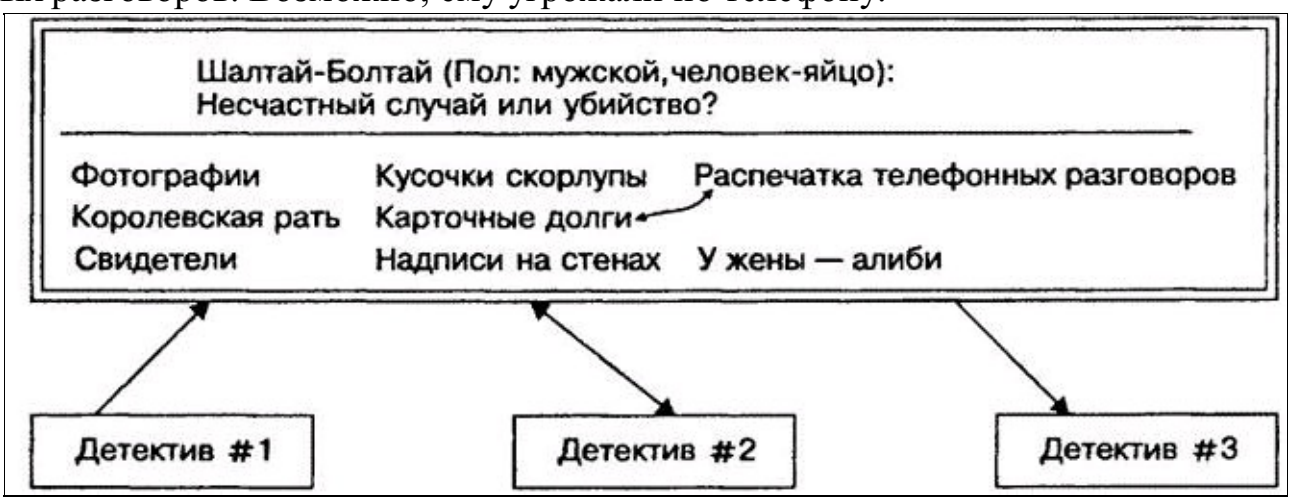
• Ни один из детективов не обязан знать о существовании какого-либо другого детектива – они лишь смотрят на доску в поисках новой информации и помещают на ней свои находки.

• Детективы могут пройти подготовку по различным дисциплинам, могут обладать различным уровнем образования и опыта и могут даже не работать на той же территории. Их объединяет желание раскрыть дело и только.

• Разные детективы могут приходить и уходить в ходе процесса, а также могут работать в различных сменах.

• На доску можно помещать все, что угодно. Это могут быть изображения, тексты, вещественные доказательства и т. д.

Рис. 5.6. Кто-то обнаружил связь между карточными долгами Шалтая и распечаткой телефонных разговоров. Возможно, ему угрожали по телефону.



Мы работали над несколькими проектами, которые включали в себя сбор распределенных данных или данных о последовательности операций. Каждый проект, решение которого строилось на основе простой модели доски объявлений, давал нам надежную метафору, с

которой мы работаем: все вышеперечисленные средства, используемые детективами, также применимы к объектам и программным модулям.

Доска объявлений позволяет полностью отделять объекты друг от друга, обеспечивая тем самым пространство, на котором потребители и производители информации могут обмениваться данными анонимно и в асинхронном режиме. Как вы могли догадаться, это также позволило уменьшить объем программ, которые нам приходилось писать.

Реализация концепции доски объявлений

Изначально доски объявлений (на основе компьютеров) разрабатывались в системах искусственного интеллекта для решения крупномасштабных и сложных задач – распознавания речи, принятии решений на основе баз знаний и т. д.

Современные распределенные системы (подобные доскам объявлений), такие как JavaSpaces и T Spaces [URL 50, URL 25], основаны на модели пар «ключ-значение», изначально пропагандировавшейся в системе Linda [CG90], где этот принцип был известен под именем "область кортежей".

При помощи этих систем можно сохранять активные объекты Java (а не только данные) на доске объявлений и извлекать их при частичном соответствии полей (через шаблоны и трафаретные символы) или с использованием подтипов. Предположим, что имеется тип Author, являющийся подтипом Person. Вы можете искать доску объявлений, содержащую объекты Person, используя шаблон Author, в котором параметру lastName присвоено значение «Shakespeare». В результате вы получите автора по имени Bill Shakespeare, а не садовника по имени Fred Shakespeare. Основные операции в системе JavaSpaces:

Название – Функция

read – Осуществляет поиск и извлечение данных из данной области.

write – Помещает некий элемент в данную область.

take – Подобен read, но также удаляет элемент из данной области.

notify – Задает вид уведомления, которое присылается при записи объекта, совпадающего с шаблоном.

Система T Spaces поддерживает аналогичный набор операций, но с другими наименованиями и несколько другой семантикой. Обе системы построены подобно базе данных; они обеспечивают элементарные операции и распределенные транзакции, гарантирующие целостность данных.

Поскольку мы можем хранить объекты, то можно использовать доску объявлений для проектирования алгоритмов, основанных на потоке объектов, а не только на данных. Любой может задать свидетелю вопросы, касающиеся расследования, поместить протокол и переместить свидетеля на другой участок доски, где он отвечал по-другому (если вы дадите и ему прочесть написанное на доске).

Большим преимуществом систем подобного типа является единственный непротиворечивый интерфейс к "доске объявлений". При построении обычного распределенного приложения вы можете затратить много времени, обрабатывая уникальные вызовы API для каждой распределенной транзакции и интеракции в системе. Проект быстро станет сущим кошмаром, если произойдет комбинаторный взрыв интерфейсов и интеракций.

Как организовать доску объявлений

Когда детективы ведут расследование крупных дел, то доска объявлений может прийти в беспорядок и найти на ней нужные данные станет сложно. Решение состоит том, чтобы разбить доску на секции и начать каким-то образом упорядочивать данные.

Различные программные системы осуществляют это разбиение по-разному; одни используют достаточно однородные зоны или группы интересов, тогда как другие используют более иерархичную древовидную структуру.

Стиль программирования под названием "доска объявлений" снимает потребность во многих интерфейсах, позволяя создавать более элегантную и последовательную систему.

Пример приложения

Предположим, что мы пишем программу для принятия и обработки заявлений на ипотечный кредит или ссуду. Законы, действующие в этой области, отличаются одиозной сложностью, и чиновникам различного уровня всегда есть что сказать по данному поводу. Кредитор обязан убедить заявителя в том, что он должен раскрыть некоторые факты и запросить определенную информацию, но не должен задавать других конкретных вопросов и т. д.

Помимо отвратительных правовых норм, нам приходится бороться со следующими проблемами.

• Порядок поступления данных никак не гарантируется. Например, выполнение запросов для проверки кредитоспособности или поиска названия требует существенных временных затрат, тогда как фамилия и адрес могут быть найдены сразу.

• Сбор данных может осуществляться разными людьми, рассеянными по разным офисам, расположенным в различных часовых поясах.

• Некоторые данные могут собираться автоматически с помощью других систем. Эти данные могут поступать в асинхронном режиме.

• И тем не менее, некоторые данные могут находится в зависимости от других данных. Например, вы не сможете начать поиск автомобиля по названию, пока не получите подтверждение права собственности или страховки.

• Поступление новых данных может вызвать появление новых вопросов и стратегии действий. Предположим, что проверка кредитоспособности заканчивается неубедительным результатом; теперь вам придется заполнить еще пять формуляров и, возможно, сдать анализ крови.

Вы можете попробовать обрабатывать всевозможные сочетания и обстоятельства, используя систему автоматизации документооборота. Существует большое число подобных систем, но они могут быть сложными и требовать интенсивной работы программистов. При изменении нормативов необходимо менять и документооборот: людям придется изменять процедуры и переписывать встроенную логику.

Доска объявлений в сочетании с механизмом правил, который включает в себя юридические требования, представляет собой изящное решение имеющих место проблем. Порядок поступления данных является несущественным параметром: регистрация некоего факта активизирует соответствующие правила. Обработка сигналов обратной связи также не представляет труда: результат действия любой совокупности правил может поместить на доску и вызвать активизацию более подходящих в данной ситуации правил.

Подсказка 43: Используйте доски объявлений для координации потоков работ

Можно использовать доску для координации неоднородных фактов и агентов, одновременно сохраняя независимость и даже изоляцию участников друг от друга.

Вы можете добиться тех же результатов, действуя и более грубыми методами, но в результате получите более хрупкую систему. Когда она сломается, даже "вся королевская конница и вся королевская рать" не смогут заставить работать вашу программу.

Другие разделы, относящиеся к данной теме:

- Преимущество простого текста
- Всего лишь визуальное представление

Вопросы для обсуждения

- Используете ли вы доски объявлений в реальности – памятные записки дома, рядом с холодильником или большие лекционные доски на работе? Что делает их эффективными? Всегда ли формат помещаемых сообщений является последовательным? Имеет ли это значение?

Упражнения

30. Будет ли уместным использование системы "доска объявлений" для приложений, указанных ниже, или нет? Почему? (Ответ см. в Приложении В.)

1. Обработка изображений. Несколько параллельных процессов захватывают фрагменты изображения, обрабатывают их и помещают обработанный фрагмент обратно.

2. Календарное планирование для групп. Есть группа людей, находящихся в разных странах, в различных часовых поясах, говорящих на разных языках и пытающихся спланировать встречу.

3. Средство мониторинга компьютерной сети. Система осуществляет сбор статистических данных о производительности сети и отчетов о неполадках. Вы хотели бы реализовать несколько программ-агентов, которые могли бы использовать эту информацию для отслеживания неисправностей в системе.

Глава 6
Пока вы пишете программу

Житейская мудрость гласит, что как только проект переходит в стадию написания текстов программ, работа становится большей частью механической, преобразующей спроектированную конструкцию в набор исполняемых операторов. Мы полагаем, что подобное отношение является единственной и самой серьезной причиной того, что многие программы уродливы, неэффективны, плохо структурированы, сложны в сопровождении и просто ошибочны.

Написание программ – не механическая процедура. В противном случае CASE-средства, с которыми специалисты связывали свои надежды в начале 80-х годов прошлого века, уже давно заменили бы программистов. Существуют решения, которые необходимо принимать ежеминутно, решения, требующие тщательного обдумывания и оценки, дающие написанной программе право на долгую, праведную и продуктивную жизнь.

Разработчики, не проявляющие активности при обдумывании своей программы, программируют в расчете на стечение обстоятельств. Программа, может быть, и работает, но этому нет определенного объяснения. В разделе "Программирование в расчете на стечение обстоятельств" мы призываем к большему участию в процессе написания программы.

Несмотря на то, что большинство составляемых нами программ выполняются быстро, иногда мы разрабатываем алгоритмы, которые способны «посадить» даже Самые быстрые процессоры. В разделе "Скорость алгоритма" обсуждаются методы оценки скорости работы программы и приводятся некоторые подсказки, предупреждающие возникновение потенциальных проблем.

Прагматики относятся критически ко всем программам, включая собственные. Мы всегда находим резервы улучшения в наших программах и конструкциях. В разделе «Реорганизация» рассматриваются методики, помогающие исправлять существующий текст программы, даже если проект находится в самом разгаре.

Всякий раз при написании текста программы необходимо помнить следующее: придет время, когда вам нужно будет ее тестировать. Сделайте так, чтобы тестирование не оказалось сложной процедурой, и вероятность того, что программа пройдет тестирование, увеличится. Эту идею мы развиваем в разделе "Программа, которую легко тестировать".

И наконец, в разделе "Злые волшебники" говорится о том, что необходимо быть осторожным с инструментальными средствами, генерирующими миллионы строк от вашего имени, если вы не понимаете сути работы этих средств.

Многие из нас в значительной степени управляют автомобилем "на автопилоте" – мы не даем явных указаний ноге, чтобы она нажала на педаль, или руке, чтобы она повернула руль, а мысленно говорим себе: "снизить скорость и повернуть направо". Но дисциплинированные водители постоянно контролируют ситуацию, отыскивают потенциальные проблемы и оказываются в нужном положении, если происходит непредвиденное. Это применимо и к написанию программ – возможно, об этом говорилось уже много раз, но хладнокровие всегда позволит вам предотвратить катастрофу.

31
Программирование в расчете на стечение обстоятельств

Случалось ли вам когда-нибудь смотреть старые черно-белые фильмы о войне? Усталый солдат осторожно выбирается из зарослей кустарника. Впереди него свободное пространство, и солдат задается вопросом: есть ли впереди мины или можно безбоязненно идти дальше? Ничто не говорит о том, что впереди минное поле, – нет ни знаков, ни колючей проволоки, ни воронок. Солдат пробует штыком грунт впереди себя и вздрагивает в ожидании взрыва. Но ничего не происходит. Какое-то время он продолжает осторожно продвигаться но полю, прощупывая грунт. В конце концов, убедившись, что проход безопасен, он распрямляется и начинает гордо маршировать вперед… навстречу смерти.

Первые поиски мин, проведенные солдатом, были безрезультатны, но ему просто повезло. Он пришел к ложному заключению, которое закончилось катастрофой.

Программисты также работают на заминированной территории. Существуют сотни ловушек, подстерегающих нас ежедневно. Помня об истории с солдатом из фильма, нам стоит опасаться ложных заключений. Необходимо избегать программирования в расчете на стечение обстоятельств, полагаясь на удачу и случайные успехи, и сделать выбор в пользу преднамеренного программирования.

Как программировать в расчете на стечение обстоятельств

Предположим, Фреду дано задание написать программу. Фред составляет некую программу, пробует ее запустить, и она вроде бы работает. Фред пишет еще один фрагмент, пробует его запустить, и снова все работает. В такой обстановке проходит еще несколько недель, но внезапно программа прекращает работать, и, потратив несколько часов на устранение дефекта, Фред все еще не знает, в чем причина. Фред может потратить много времени, копаясь с этим фрагментом, без перспективы на восстановление работы программы. И что бы он ни делал, кажется, что программа никогда не будет работать правильно.

Фред не знает, почему программа сбоит, потому что не знает, почему она работала вначале. Она лишь казалась работающей в условиях ограниченного «тестирования», которое проводил Фред, но это было лишь стечением обстоятельств. Находясь в плену ложной уверенности, Фред впал в забытье. Большинству интеллектуалов знаком этот образ Фреда, но мы знаем его лучше. Мы ведь не полагаемся на стечение обстоятельств, не так ли?

Впрочем, иногда полагаемся. Порой легко спутать счастливый случай с целенаправленным планированием. Рассмотрим несколько примеров.

Случайная реализация

Случайная реализация – это то, что происходит просто потому, что программа написана именно так, как она написана. Вы перестаете полагаться на недокументированную ошибку или граничные условия.

Предположим, что вы вызываете подпрограмму с неверными данными. Подпрограмма откликается определенным образом, и ваша программа основывается на этом отклике. Но у автора даже и в мыслях не было, что программа будет работать подобным образом, – это даже не рассматривалось. Если подпрограмма «исправляется», то основная программа может

нарушиться. В самом крайнем случае вызываемая подпрограмма даже не предназначена для того, чего вы от нее ждете, но вроде бы она работает нормально. Вызов каких-либо элементов неправильным образом или в неверном контексте является связанной проблемой.

```
paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);
```

Похоже, что Фред предпринимает отчаянные попытки вывести что-то на экран. Но эти подпрограммы не предназначены для того, чтобы к ним обращались таким способом; хотя они кажутся работающими, в действительности это лишь стечение обстоятельств.

Чтобы не получить новых ударов, когда компонент все-таки нарисован, Фред не пытается вернуться назад и устранить поддельные запросы. "Сейчас она работает, оставим все как есть…".

Подобные размышления могут ввести вас в заблуждение. Зачем рисковать, портить то, что работает? Так можно думать по нескольким причинам:

• Программа действительно может не работать, она может лишь казаться работающей.

• Граничное условие, на которое вы полагаетесь, может быть лишь частным случаем. В различных обстоятельствах (например, при ином экранном разрешении) программа может вести себя по-разному.

• Недокументированное поведение может измениться с выпуском новой версии библиотеки.

• Дополнительные и необязательные вызовы замедляют работу программы.

• Дополнительные вызовы также увеличивают риск привнесения новых дефектов, связанных с этим вызовами.

При написании программы, вызываемой другими разработчиками, полезными могут оказаться базовые принципы четкой модуляризации и скрытия реализации за несложными, четко документированными интерфейсами. Четко определенный контракт (см. "Проектирование по контракту") может устранить недоразумения.

Для вызываемых вами подпрограмм полагайтесь только на документированное поведение. Если по какой-то причине вы не можете сделать этого, то четко документируйте ваше предположение.

Случайный контекст

Вы также можете встретиться со "случайным контекстом". Предположим, вы пишете сервисный модуль. Поскольку в данное время вы пишете программу для графической среды, должен ли модуль полагаться на существующий графический интерфейс? Полагаетесь ли вы на англоязычных пользователей? На грамотных пользователей? Полагаетесь ли вы еще на какой-то контекст, наличие которого не гарантируется?

Неявные предположения

Совпадения могут вводить в заблуждение на всех уровнях – от генерации требований до

тестирования. Тестирование особенно чревато наличием ложных причинных связей и случайным совпадением результатов. Легко предположить, что А вызывает У, но, как сказано в разделе «Отладка» не предполагайте это, а доказывайте.

На всех уровнях люди работают, держа многие предположения в голове, но они редко документируются и часто вызывают противоречия между разработчиками. Предположения, не основанные на известных фактах, способны отравить любые проекты.

Подсказка 44: Не пишите программы в расчете на стечение обстоятельств

Преднамеренное программирование

Мы хотели бы тратить меньше времени на придание нашим программам компактности, как можно раньше перехватывая и устраняя ошибки, возникающие в ходе разработки, а для начала допускать меньшее число ошибок. Этот принцип приносит пользу, если мы способны программировать преднамеренно:

• Всегда отдавайте себе отчет в том, что вы делаете. Программист Фред постепенно терял контроль над происходящим, пока не сварился сам, подобно лягушке из раздела "Суп из камней и сварившиеся лягушки".

• Не пишите программ вслепую. Попытка написать приложение, которое вы до конца не понимаете, или использовать технологию, с которой вы не знакомы, становится поводом к тому, что вы будете введены в заблуждение случайными совпадениями.

• Действуйте исходя из плана, неважно, где он составлен – у вас в голове, на кухонной салфетке или на огромной «простыне», полученной с помощью CASE-средств.

• Полагайтесь только на надежные предметы. Не вводите себя в зависимость от случаев или предположений. Если вы не можете понять, в чем состоит различие при специфических обстоятельствах, предполагайте худшее.

• Документируйте ваши предположения. Раздел "Проектирование по контракту" поможет прояснить ваши предположения в вашей же голове, а также передать их другим людям.

• Тестируйте не только вашу программу, но и ваши предположения. Не гадайте, попробуйте осуществить это на деле. Напишите программу контроля для проверки ваших предположений (см. "Программирование утверждений"). Если ваше предположение верно, то вы улучшили документирование вашей программы. Если вы обнаружили, что предположение ошибочно, тогда считайте, что вам повезло.

• Определите приоритеты в своей работе. Уделите время аспектам, представляющим важность; скорее всего, они окажутся непростыми. При отсутствии надлежащих фундаментальных принципов или инфраструктуры все блестящие «бантики» будут просто неуместны.

• Не будьте рабами прошлого. Не позволяйте существующей программе диктовать свою волю той программе, за которой будущее. Если программа устаревает, она может быть полностью заменена. И даже в пределах одной программы не позволяйте уже сделанному сдерживать то, что идет за ним, – будьте готовы к реорганизации (см. "Реорганизация"). Это решение может повлиять на график выполнения проекта. Мы полагаем, что это воздействие будет меньше той цены, которую придется платить за отсутствие изменений [37].

Поэтому, если в следующий раз что-то начинает работать, но вы не знаете, почему это

происходит, убедитесь, что это не является стечением обстоятельств

Другие разделы, относящиеся к данной теме:

- Суп из камней и сварившиеся лягушки
- Отладка
- Проектирование по контракту
- Программирование утверждений
- Временное связывание
- Реорганизация
- Все эти сочинения

Упражнения

31. Найдите совпадения в представленном фрагменте программы на языке C. Предположим, что этот фрагмент находится глубоко в недрах библиотечной подпрограммы. (Ответ см. в Приложении B.)

```
fprintf(stderr, "Error, continue?");
gets(buf);
```

32. Этот фрагмент программы на языке C мог работать в течение какого-то времени на некоторых машинах. Затем он переставал работать. В чем ошибка? (Ответ см. в Приложении B.)

```
/* Truncate string to its iast maxlen chars */
void string_tail(char *string, int maxlen) {
  int len = strlen(string);
  if (len > maxlen) {
    strcpy(string, string+(len – maxlen));
  }
}
```

33. Эта программа входит в состав универсального пакета трассировки Java. Функция записывает строки в файл журнала. Она проходит модульное тестирование, но дает сбой при попытке ее применения одним из разработчиков программ для сети Интернет. На какое стечение обстоятельств полагается эта программа? (Ответ см. в Приложении B.)

```
public static void debug(String s) throws IOException {
FileWriter fw = new FileWriter("debug.log");
fw.write(s);
fw.flush();
fw.close();
}
```

32
Скорость алгоритма

В разделе «Оценка» говорилось об оценке того, сколько времени потребуется, чтобы пройти несколько городских кварталов, и сколько времени нужно для завершения проекта. Однако существует и другой вид оценок, который прагматики применяют практически ежедневно: оценка ресурсов, используемых алгоритмами, – времени, работы процессора, объема памяти и т. д.

Зачастую этот вид оценки является решающим. Если вы можете сделать что-либо двумя способами, то какой из них стоит выбрать? Если вам известно время выполнения программы при наличии 1000 записей, то как оно изменится при наличии 1000000 записей? Какая часть программы нуждается в оптимизации?

Оказывается, что во многих случаях на подобные вопросы можно ответить, пользуясь здравым смыслом, некоторым анализом и методикой записи приближений, которая называется "О-большое".

Что подразумевается под оценкой алгоритмов?

Большинство нетривиальных алгоритмов обрабатывают некий вид переменных входных массивов, они выполняют сортировку n строк, обращение матрицы размером m*n или расшифровку сообщения с n-битовым ключом. Обычно объем входных данных оказывает влияние на алгоритм: чем больше этот объем, тем больше время выполнения алгоритма или объем используемой памяти.

Если бы эта зависимость всегда была линейной (т. е. время возрастало бы прямо пропорционально значению n), то этот раздел можно было бы и пропустить. Однако наиболее важные алгоритмы не являются линейными. Хорошая новость: многие алгоритмы являются сублинейными. Например, в алгоритме двоичного поиска при нахождении соответствия вовсе не обязательно рассматривать подряд всех кандидатов. А теперь плохая новость: другие алгоритмы отличаются существенно худшими линейными свойствами; время их выполнения или требования к объему памяти возрастают намного быстрее, чем значение n. Если для обработки десяти элементов алгоритму требуется минута, то для обработки ста элементов потребуется целая жизнь.

При написании любых программ, содержащих циклы или рекурсивные вызовы, мы подсознательно проверяем требования, предъявляемые ко времени выполнения и объему памяти. Это редко является формальным процессом, скорее, оперативным подтверждением наличия здравого смысла в том, что мы делаем в определенных обстоятельствах. Но иногда мы оказываемся в ситуации, когда нам приходится проводить более детальный анализ. В этом случае весьма полезной оказывается система обозначений "O()" ("О-большое").

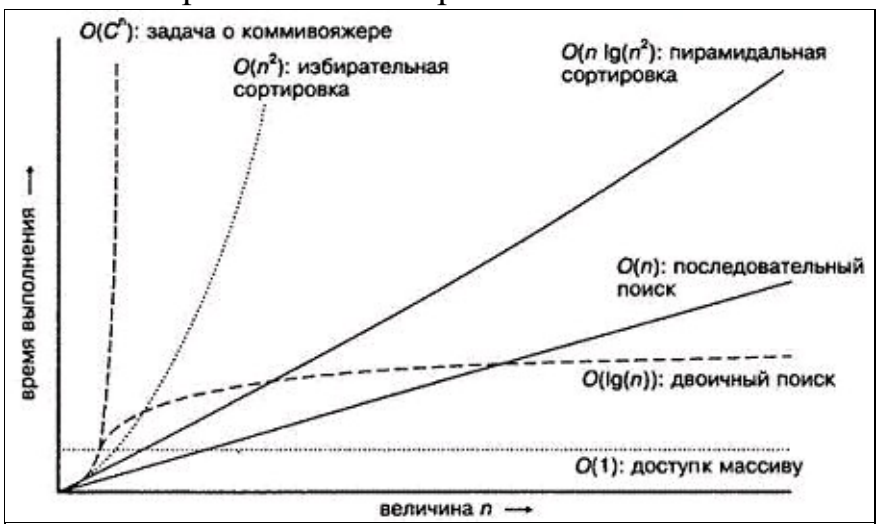
Система обозначений O()

Система O() представляет собой математический способ обозначения приближений. Если мы указываем, что некая программа осуществляет сортировку n записей за время O(n^2), то это просто означает, что максимальное время выполнения программы будет изменяться пропорционально n^2. При удвоении числа записей время возрастет примерно в четыре раза. O()

можно рассматривать как порядок величины. Система обозначений O() определяет верхнюю границу величины измеряемого параметра (время, объем памяти, и т. д.). Если мы говорим, что некая функция занимает время O(n^2), то под этим понимается, что верхняя граница интервала времени, необходимого для ее выполнения, возрастает не быстрее n^2. Иногда мы встречаемся с довольно сложными функциями O(), и поскольку именно член высшего порядка будет определять значение с ростом n, то обычно все члены низшего порядка удаляются, чтобы не мешать постоянным коэффициентам умножения. O(n^2/2+3n) означает то же самое, что и O(n^2/2), которое, в свою очередь, является эквивалентом O(n^2). В этом и состоит недостаток системы обозначений O() – один алгоритм O(n^2) может быть быстрее другого алгоритма O(n^2) в тысячу раз, но из обозначений вы этого не поймете.

На рисунке 6.1 показано несколько общих обозначений O(), с которым вы можете встретиться, и график, на котором сравнивается время выполнения алгоритмов в каждой категории. Из него ясно, что все начинает быстро выходить из-под контроля, как только мы переходим через O(n^2).

Рис. 6.1. Время выполнения различных алгоритмов



Некоторые универсальные обозначения O-большое

O(1) Постоянная зависимость (обращение к элементу массива, простые операторы)

O(lg(n)) Логарифмическая зависимость (двоичный поиск) [lg(n) – краткое обозначение log2(n)]

O(n) Линейная зависимость (последовательный поиск)

O(n lg(n)) Эта зависимость линейной, но не намного (среднее время быстрой сортировки, пирамидальной сортировки)

O(n^2) Квадратичная зависимость (выборочная сортировка и сортировка включения)

O(n^3) Кубическая зависимость (перемножение двух матриц размером n*n)

O(C^n) Экспоненциальная зависимость (задача о коммивояжере, разбиение набора)

Предположим, что у вас есть программа, обрабатывающая 100 записей за 1 сек. Сколько времени ей потребуется для обработки 1000 записей? Если ваша программа является O(1), то это время остается равным 1 сек. Если она является O(lg(n)), то для обработки потребуется около 3 сек. При O(n) время обработки линейно возрастает до 10 сек., а при O(nlg(n)) составит примерно 33 сек. Если вам не повезло и ваша программа является O(n^2), то можете отдохнуть в течение 100 сек., пока она не сделает свое дело. Ну а в том случае, если вы используете экспоненциальный алгоритм O(2^n), можете заварить чашечку кофе – программа завершит свою

работу примерно через 10263 года. В общем, хотелось бы знать, как происходит конец света.

Система обозначений O() не применяется только к временным параметрам; ее можно использовать для представления других ресурсов, требуемых неким алгоритмом. Например, она часто является полезной при моделировании расхода памяти (см. упражнение 35).

Оценка с точки зрения здравого смысла

Можно оценить порядок многих базовых алгоритмов с точки зрения здравого смысла.

• **Простые циклы.** Если простой цикл выполняется от 1 до n, то алгоритм, скорее всего, является O(n) – время находится в линейной зависимости от n. Примерами этого являются исчерпывающий поиск, поиск максимального элемента в массиве и генерация контрольной суммы.

• **Вложенные циклы.** Если вы помещаете один цикл в другой, то ваш алгоритм становится O(m*n), где m и n – пределы этих двух циклов. Обычно это свойственно простым алгоритмам сортировки, типа пузырьковой сортировки, где внешний цикл поочередно просматривает каждый элемент массива, а внутренний цикл определяет местонахождение этого элемента в результирующем массиве. Подобные алгоритмы сортировки чаще всего стремятся к O(n^2).

• **Алгоритм двоичного поиска.** Если алгоритм делит пополам набор элементов, который он рассматривает всякий раз в цикле, то скорее всего он логарифмический O(lg(n)) (см. упражнение 37). Двоичный поиск в упорядоченном списке, обход двоичного дерева и поиск первого установленного бита в машинном слове могут быть O(lg(n)).

• **Разделяй и властвуй.** Алгоритмы, разбивающие входные данные на разделы, работающие независимо с двумя половинами и затем комбинирующие конечный результат, могут представлять собой O(nlg(n)). Классическим примером является алгоритм быстрой сортировки, который делит входной массив пополам и затем проводит рекурсивную сортировку в каждой из половин. Хотя технически он и является O(n^2), поскольку его поведение ухудшается при обработке упорядоченных данных, но среднее время быстрой сортировки составляет O(nlg(n)).

• **Комбинаторика.** При использовании алгоритмов в решении любых задач, связанных с перестановкой, время их выполнения может выйти из-под контроля.

Это происходит потому, что задачи о перестановке включают вычисления факториалов (существует 5! = 5*4*3*2*1 = 120 перестановок цифр от 1 до 5). Возьмем за основу время выполнения комбинаторного алгоритма для пяти элементов; для шести элементов времени потребуется в шесть раз больше, а для семи – в 42. Примерами этого являются алгоритмы решения многих известных сложных задач – о коммивояжере, об оптимальной упаковке предметов в контейнер, о разделении набора чисел таким образом, что сумма каждого отдельного набора одинакова и т. д. Во многих случаях для сокращения времени выполнения алгоритмов данного типа в определенных прикладных областях используются эвристические подходы.

Скорость алгоритма на практике

Маловероятно, что в своей профессиональной карьере вам придется тратить много времени на написание программ сортировки. Эти программы, входящие в стандартные библиотеки, наверняка без особых усилий превзойдут написанное вами. Но основные типы алгоритмов, описанные выше, будут время от времени всплывать на поверхность. Во всех случаях, когда вы пишете простой цикл, знайте, что имеете дело с алгоритмом O(n). Если же этот цикл содержит

внутренний цикл, то речь идет о O(m*n). Вы обязаны задаться вопросом: а насколько велики эти значения? Если эти значения ограничены сверху, то вы можете представить, сколько времени потребуется на выполнение программы. Если эти цифры зависят от внешних факторов (наподобие количества записей в запускаемом на ночь пакете программ или количества фамилий в списке персоналий), то стоит остановиться и изучить влияние больших чисел на время выполнения программы или объемы необходимой памяти.

Подсказка 45: Оцените порядок ваших алгоритмов

Существует несколько подходов, которыми вы можете воспользоваться при решении потенциально возникающих проблем. Если есть алгоритм, являющийся O(n^2), попробуйте действовать по принципу "разделяй и властвуй", что может уменьшить время выполнения до O(nlg(n)).

Если вы не уверены в том, что ваша программа будет выполняться в течение определенного времени, или в том, что она затребует определенный объем памяти, попытайтесь запустить ее, варьируя количество обрабатываемых записей или другие параметры, способные оказать воздействие на время выполнения программы. На основе полученных результатов постройте график и получите представление о форме кривой. Изгибается ли она кверху, представляет ли собой прямую линию или сглаживается с увеличением размера входного массива данных? Представление об этом можно получить, исходя из трех или четырех точек.

Стоит рассмотреть и то, что происходит в самой программе. При малых значениях n простой цикл O(n^2) может работать намного лучше, чем сложный O(nlg(n)), особенно если последний содержит ресурсоемкий внутренний цикл.

Говоря о теории, не стоит забывать и о практических соображениях. При работе с небольшими массивами входных данных может показаться, что время выполнения возрастает линейно. Но если программа обрабатывает миллионы записей, то внезапно время выполнения резко увеличивается, по мере того как система начинает «буксовать». При проведении тестирования программы сортировки со случайными входными ключами вы можете удивиться ее работе с упорядоченным входным массивом. Прагматики стараются обеспечивать как теоретическую, так и практическую базу. После всех проведенных оценок единственной определяемой временной характеристикой является скорость выполнения вашей программы в реальных условиях эксплуатации и с реальными данными [38]. Из этого следует следующая подсказка.

Подсказка 46: Проверяйте ваши оценки

Если сложно точно определить время, воспользуйтесь программами оптимизации, чтобы подсчитать, сколько раз выполнялся алгоритм, и постройте зависимость этого количества от размера входного массива данных.

Лучшее – враг хорошего

При выборе подходящего алгоритма также необходимо придерживаться прагматического подхода – самые быстрые алгоритмы не обязательно являются наилучшими для конкретного случая. При небольшом входном массиве «прямолинейная» сортировка со вставкой будет работать так же хорошо, как и алгоритм быстрой сортировки, и потребует меньше времени на написание и отладку. Необходимо соблюдать осторожность, если выбранный вами алгоритм отличается высокими затратами на установку. При работе с небольшими массивами эта дорогостоящая установка может свести на нет преимущество в скорости выполнения и сделать алгоритм нерентабельным.

Кроме того, необходимо опасаться преждевременной оптимизации. Перед тем как потратить ваше драгоценное время на улучшение алгоритма, всегда есть смысл убедиться, что он действительно является "узким местом".

Другие разделы, относящиеся к данной теме:

• Оценка

Вопросы для обсуждения

• Каждый разработчик должен обладать чутьем на проектирование и анализ алгоритмов. По данному предмету Роберт Седжвик написал серию доступных книг ([Sed83, SF96, Sed92]и др.). Мы рекомендуем пополнить вашу библиотеку одной из этих книг и обязательно прочесть ее.

• Те, кто интересуется данным предметом более глубоко (по сравнению с его подачей в книге Седжвика), могут прочесть каноническую серию книг Дональда Кнута "Искусство программирования", в которых анализируются разнообразные алгоритмы [Knu97a, Knu97b, Ктш98].

• В упражнении 34 рассматривается сортировка массивов, состоящих из чисел типа "длинное целое". Как скажутся на сортировке усложнение ключей и издержки на их сравнение? Оказывает ли структура ключей влияние на эффективность работы алгоритмов сортировки, словом, является ли самый быстрый алгоритм сортировки таковым во всех случаях?

Упражнения

34. Авторы книги составили набор простых программ сортировки, которые можно загрузить с их Интернет-сайта (www.pragmaticprogrammer.com). Прогоните эти программы на разных компьютерах, имеющихся в вашем распоряжении. Соответствуют ли полученные вами данные ожидаемым кривым? Какие заключения можно сделать об относительных скоростях ваших машин? Каково влияние различных установочных параметров компиляторов? Является ли поразрядная сортировка действительно линейной? (Ответ см. в Приложении В.)

35. Приведенная ниже подпрограмма выводит на печать содержимое двоичного дерева. Предполагая, что дерево сбалансировано, какой (примерно) объем стека будет использоваться подпрограммой для вывода на печать дерева, состоящего из 1000000 элементов? (Предполагается, что вызовы подпрограммы не оказывают существенной нагрузки на стек). (Ответ см. в Приложении В.)

```
void printTree(const Node *node) {
```

```
char buffer[1000];
 if (node) {
   printTree(node->left);
   getNodeAsString(node, buffer);
   puts(buffer);
   printTree(node->right);
 }
}
```

36. Существует ли способ уменьшить потребность подпрограммы, описанной в упражнении 35, в ресурсах стека (помимо уменьшения размера буфера)? (Ответ см. в Приложении В.)

37. В разделе "Оценка с точки зрения здравого смысла" утверждается, что алгоритм двоичного поиска является O(lg(n)). Можно ли это доказать? (Ответ см. в Приложении В.)

33
Реорганизация

Как изменилось и увяло все, что окружает меня…

Г.Ф. Лайт, Пребудь со мной

По мере развития программы возникает необходимость в переосмыслении ранее принятых решений и переработки отдельных фрагментов текста программы. Этот процесс абсолютно естественен. Программа нуждается в эволюции, она не является статическим объектом.

К сожалению, наиболее распространенной метафорой разработки программного обеспечения является строительство здания (Б. Мейер [Мey97Ь] использует термин "Software Construction" – букв.: строительство программ – Прим. пер.). Но использование термина «строительство» в качестве определяющей метафоры подразумевает наличие следующих стадий:

1. Архитектор готовит чертежи на кальке.
2. Фирмы-подрядчики роют котлован под фундамент, возводят наземную часть, проводят электричество, монтируют водопровод и канализацию и осуществляют отделочные работы.
3. Арендаторы въезжают в дом и с этого времени живут-поживают, лишь иногда обращаясь в домоуправление с просьбой устранить возникшие неисправности.

Программное обеспечение работает несколько по-иному. В отличие от строительства, написание программ ближе к садоводству, оно ближе к живой природе, чем к бетонным конструкциям. Вы высаживаете в саду множество растений согласно первоначальному плану и условиям. Некоторые растения разрастаются, другим же уготована компостная яма. Вы можете пересаживать растения друг относительно друга, чтобы извлечь пользу из взаимодействия света и тени, ветра и дождя. Переросшие растения разрубают или обрезают, растения определенного цвета пересаживают на другие участки, где они становятся более приятными глазу с точки зрения эстетики. Вы выпалываете сорняки и подкармливаете растения, которые нуждаются в дополнительном питании. Вы постоянно следите за состоянием сада и при необходимости вносите изменения (в почву, растения, общий план).

Для бизнесменов понятнее метафора строительства здания, она более научна по сравнению с садоводством, она воспроизводима, в управлении есть жесткая иерархия подотчетности и т. д. Но мы не занимаемся строительством небоскребов – можем выйти за рамки физики и реального мира.

Метафора садоводства намного ближе к реальности разработки программного обеспечения. Возможно, некая программа переросла себя или пытается осуществить слишком много – ее необходимо разбить на две. Все, что не получается в соответствии с планом, подлежит прополке или обрезке.

Переписывание, переработка и перепланирование текста программы описывается общим термином "реорганизация".

Когда осуществлять реорганизацию?

Если вы встречаете на своем пути камень преткновения, поскольку текст программы никуда не годится, замечаете, что два объекта стали несовместимы друг с другом, или же нечто другое, что задевает вас своей «неправильностью», не стесняйтесь вносить изменения. Другого времени, кроме настоящего, не существует. Программу можно считать пригодной для

реорганизации при наличии одного из указанных ниже условий:

• **Дублирование.** Вы обнаружили нарушение принципа DRY (см. "Пороки дублирования").

• **Неортогональность конструкции.** Вы обнаружили некий фрагмент программы или конструкцию, которой можно придать большую ортогональность (см. "Ортогональность").

• **Устаревшие знания.** Все изменяется, требования варьируются, и ваши знания о проблеме расширяются. Программа должна соответствовать новому уровню знаний.

• **Рабочие характеристики.** Для улучшения характеристик программы вам необходимо перенести функциональную возможность из одной части системы в другую.

Реорганизация программы, т. е. перемещение функциональной возможности и изменение ранее принятых решений – это упражнение в обезболивании. Скажем сразу – изменение исходного текста программы может быть весьма болезненной процедурой: она уже почти работала, а теперь ее разрывают в клочья. Многие разработчики крайне неохотно соглашаются «вспарывать» программу лишь на том основании, что она работает не совсем правильно.

Осложнения в реальном мире

Итак, вы идете к вашему шефу или заказчику и говорите: "Эта программа работает, но для ее реорганизации мне нужна еще неделя".

Они скажут вам… впрочем, это непечатное выражение.

На жесткие временные рамки часто ссылаются, оправдывая отсутствие реорганизации. Но это оправдание не должно становиться нормой: если вы не сможете провести реорганизацию сейчас, то позже (когда придется принимать во внимание большее число зависимостей) на устранение возникшей проблемы потребуется намного больше времени. А будет ли у вас это время? У нас – точно не будет.

Попробуйте объяснить этот принцип вашему шефу, пользуясь аналогией с медициной: рассматривайте программу, нуждающуюся в реорганизации, как «опухоль». Чтобы удалить ее, требуется хирургическое вмешательство. Вы можете начать сразу и извлечь ее, пока она небольшая. Но если вы будете ждать, пока она вырастет и распространится, то ее удаление станет более дорогой и опасной процедурой. Подождите еще, и вы можете потерять пациента окончательно.

Подсказка 47: Реорганизация должна проводиться часто и как можно раньше

Следите за всем, что требует реорганизации. Если вы не можете провести реорганизацию чего-либо прямо сейчас, удостоверьтесь, что она стоит в вашем плане. Убедитесь, что пользователи программы, над которой производится реорганизация, знают о запланированной процедуре и о том, как она может повлиять на их работу.

Как производится реорганизация?

Реорганизация появилась в среде программистов, работающих с языком Smalltalk, и начала, вкупе с другими модными поветриями (например, шаблоны конструкций), завоевывать все более широкую аудиторию. Но это еще малоизвестная тема, по ней опубликовано не так много работ. Первая большая монография о реорганизации ([FBB+99], а также [URL 47]) вышла

одновременно с данной книгой.

Суть реорганизации заключается в перепланировке. Все спроектированное вами или другими членами вашей команды может быть переделано в свете новых фактов, более глубокого понимания, изменения требований и т. д. Но если вы предадите забвению огромные фрагменты программы, то окажетесь в худшем положении, чем в начале работы по реорганизации.

Ясно, что реорганизация представляет собой род деятельности, которая должна осуществляться медленно, преднамеренно и осторожно. Мартин Фаулер предлагает ряд простых подсказок – как провести реорганизацию, чтобы это не принесло больше вреда, чем пользы (см. врезку на стр. 30 в книге [FS97]):

1. Не пытайтесь одновременно производить реорганизацию и добавлять функциональные возможности.

2. Перед тем как начинать реорганизацию, убедитесь, что тестирование прошло успешно. Проводите тестирование как можно чаще. В этом случае вы сразу увидите нарушение, которое было вызвано внесенными изменениями.

Автоматическая реорганизация

Исторически сложилось так, что пользователи Smalltalk всегда применяли средство просмотра классов как неотъемлемую часть интегрированной среды разработчика. В отличие от web-браузеров, средства просмотра классов позволяют пользователям перемещаться по иерархиям и методам класса и проверять их.

Обычно средства просмотра классов позволяют редактировать текст программы, создавать новые методы, классы и т. д. Следующей вариацией на эту тему является браузер реорганизации.

Этот браузер может в полуавтоматическом режиме проводить операции, обычные при реорганизации: разбивать длинную подпрограмму на несколько более коротких, автоматически перенося изменения на имена методов и переменных, а также осуществлять операцию "буксировки и перетаскивания", что помогает в перемещении текста программы и т. д.

Во время написания данной книги этой технологии еще предстояло выйти за пределы мира Smalltalk, но скорее всего она начнет меняться с той же скоростью, что и язык Java, – быстро. В то же время исторический браузер реорганизации Smalltalk можно отыскать в Интернете [URL 20].

3. Двигайтесь обдуманно и не спеша: переместите поле из одного класса в другой, объедините два подобных метода в суперкласс. Часто при реорганизации вносится много локальных изменений, которые приводят к серьезным сдвигам. Если вы двигаетесь без спешки и проводите тестирование после каждого шага, вы избежите длительной процедуры отладки.

На данном уровне тестирование будет обсуждаться в разделе "Программа, которую легко тестировать", тестирование на более высоком уровне – в разделе "Безжалостное тестирование"), но мнение г-на Фаулера о тщательном регрессионном тестировании является ключом к надежной реорганизации.

Также весьма полезно удостовериться в том, что серьезные изменения в некоем модуле, такие как изменения его интерфейса или его функциональной возможности неподобающим способом, приведут к нарушению процесса сборки. Это означает, что прежние клиенты этой программы не смогут пройти компиляцию. Тогда вы можете отыскать старых клиентов и внести необходимые изменения, чтобы осовременить их.

Поэтому в следующий раз, когда вам попадется фрагмент программы, который не совсем

такой, каким ему надлежит быть, исправьте и его, и все то, что от него зависит. Научитесь управлять этой головной болью: если она досаждает вам сейчас, то потом будет досаждать еще больше, у вас есть шанс устранить ее совсем. Помните уроки, полученные в разделе "Энтропия в программах": не живите с разбитыми окнами.

Другие разделы, относящиеся к данной теме:

- Мой исходный текст съел кот Мурзик
- Энтропия в программах
- Суп из камней и сварившиеся лягушки
- Пороки дублирования
- Ортогональность
- Программирование в расчете на стечение обстоятельств
- Программа, которую легко тестировать
- Безжалостное тестирование

Упражнения

38. По всей вероятности, за последние годы представленная ниже программа переписывалась несколько раз, но эти изменения никак не способствовали улучшению ее структуры. Проведите ее реорганизацию. (Ответ см. в Приложении В.)

```
if (state==TEXAS) {
rate=TX.RATE;
amt=base * TX_RATE;
calc=2*basis(amt) + extra(amt)*1.05;
}
else if ((state==OHIO) || (state==MAINE)) {
rate=(state==OHIO) ? OH_RATE : MN_RATE;
amt=base*rate;
calc=2*basis(amt) + extra(amt)*1.05;
if (state==OHIO)
  points = 2;
}
else {
rate=1;
amt=base;
calc=2*basis(amt) + extra(amt)*1.05;
}
```

39. Класс Java, представленный ниже, нуждается в поддержке дополнительных форм. Произведите реорганизацию этого класса, чтобы подготовить его к этим дополнениям. (Ответ см. в Приложении В.)

```
public class Shape {
public static final int SQUARE = 1;
public static final int CIRCLE = 2;
```

```
public static final int RIGHTTRIANGLE = 3;
private int shapeType;
private double size;
public Shape(int shapeType, double size) {
   this.shapeType = shapeType;
   this.size = size;
}
//… другие методы…
public double area() {
 switch (shapeType) {
  case SQUARE: return size*size;
  case CIRCLE: return Math.PI*size*size/4.0;
  case RIGHT TRIANGLE: return size*size/2.0;
}
return 0;
}
```

40. Данная программа на языке Java представляет собой часть некоего скелета, который будет использоваться во всем вашем проекте. Произведите реорганизацию этой программы, чтобы сделать ее более общей и упростить ее расширение в будущем. (Ответ см. в Приложении В.)

```
public class Window {
  public Window(int width, int height) {…}
  public void setSize(int width, int height) {…}
  public boolean overiaps(Window w) {…}
  public int getArea() {…}
```

34
Программа, которую легко тестировать

Термин "программная интегральная схема" является метафорой, брошенной в ходе дискуссии о многократном использовании и компонентно-ориентированной разработке [39]. Идея заключается в том, что программные компоненты должны объединяться так же, как это происходит с чипами интегральной схемы. Этот подход срабатывает только в том случае, если известно, что используемые компоненты надежны.

Чипы предназначены душ тестирования не только на предприятии-изготовителе, не только при сборке, но и в сфере их применения. Более сложные чипы и системы могут снабжаться полномасштабными средствами самотестирования, которые осуществляют внутреннюю диагностику на базовом уровне, или тестовым стендом с комплектом измерительных кабелей инициирующим подачу тестовых входных сигналов и снимающим ответную информацию с чипа.

То же самое можно осуществить и с программным обеспечением. Подобно нашим коллегам, работающим с «железом», нам приходится с самого начала встраивать средства тестирования в программы и тщательно тестировать каждый фрагмент, перед тем как предпринять попытку их объединения.

Модульное тестирование

Тестирование аппаратных средств на уровне чипа отдаленно напоминает модульное тестирование программного обеспечения – тестируется каждый модуль по отдельности для проверки его поведения. Мы можем лучше представить себе, какова будет реакция модуля на внешний мир, если проведем его тщательное тестирование в контролируемых (и даже искусственных) условиях.

Модульный программный тест – это программа, испытывающая работу модуля. Обычно модульный тест задает некую искусственную среду, затем осуществляется вызов подпрограмм из проверяемого модуля. Потом происходит проверка полученных результатов, они сравниваются с известными величинами или с результатами предыдущих прогонов той же самой программы тестирования (регрессионное тестирование).

Когда мы объединим наши "программные интегральные схемы" в единую систему, мы будем уверены, что ее отдельные части работают предсказуемо, а затем можем применить те же средства модульного тестирования при проверке системы в целом. О подобном крупномасштабном тестировании речь идет в разделе "Безжалостное тестирование".

Но прежде чем выйти на этот уровень, необходимо решить, а что же мы будем тестировать на уровне блоков. Обычно программисты задают несколько случайных массивов данных и считают, что они провели тестирование. Но это можно сделать намного лучше, если использовать идеи, связанные с "программированием по контракту".

Тестирование в рамках контракта

Мы рассматриваем модульное тестирование, как тестирование исходя из контракта (см. "Проектирование по контракту"). Нам бы хотелось написать процедуры тестирования, гарантирующие, что данный модуль соблюдает соответствующий контракт. При этом

выясняются два момента: отвечает ли программа условиям контракта, и означает ли контракт на самом деле то, что мы о нем думаем. Мы хотим проверить, обладает ли модуль функциональными характеристиками, которые в нем заложены, используя разнообразные тестовые процедуры и граничные условия.

Что это означает на практике? Рассмотрим подпрограмму извлечения квадратного корня, с которой мы впервые встретились в разделе "ППК и аварийное завершение работы программы". Ее контракт довольно прост:

```
require:
  argument >=0
ensure:
 abs((result*result)–argument) < epsilon
```

Он указывает на моменты, нуждающиеся в проверке:

• Передать отрицательный аргумент и удостовериться в том, что он отклонен

• Передать аргумент, равный нулю, и удостовериться в том, что он принят (это граничное значение)

• Передать значение в интервале от нуля до максимально выражаемого параметра и проверить, что разность между квадратом результата и исходным аргументом меньше некоторой величины "epsilon"

Вооружась этим контрактом и полагая, что наша программа осуществляет собственную проверку предусловий и постусловий, можно записать базовый тестовый сценарий для проверки функции извлечения квадратного корня.

```
public void testValue(double num, double expected) {
double result = 0.0;
try {    // We may throw a
result = mySqrt(num); // precondition exception
}
catch (Throwable e) {
if (num<0.0) // If input is «0, then
  Return; // we're expecting the
Else // exception, otherwise
  Assert(false); // force a test failure
}
assert(Math.abs(expected-result)<epsilon);
}
```

Затем мы можем вызвать эту подпрограмму, чтобы проверить нашу функцию излечения квадратного корня:

```
TestValue(-4.0, 0.0);
TestValue(0.0, 0.0);
TestValue(2.0, 1.4142135624);
TestValue(64.0, 8.0);
TestValue(1.0e7, 3162.2776602);
```

Это весьма простая процедура тестирования; в реальном мире любой нетривиальный модуль скорее всего будет зависеть от ряда других модулей, поэтому, может быть, есть смысл протестировать их сочетание?

Предположим, есть модуль A, использующий модули LinkedList и Sort. Мы осуществляем тестирование в следующем порядке:

1. Полностью тестируем контракт модуля LinkedList.

2. Полностью тестируем контракт модуля Sort.

3. Тестируем контракт модуля A, который полагается на другие контракты, но не раскрывает их напрямую.

При этом способе тестирования вы вначале обязаны проводить тестирование подкомпонентов.

Если модули LinkedList и Sort успешно прошли тестирование, а модуль A испытания не прошел, мы можем быть вполне уверены, что проблема заключается в модуле A или в том, как модуль A использует один из подкомпонентов. Эта методика способствует уменьшению трудоемкости процесса отладки: можно быстро сосредоточиться на вероятном источнике проблем в пределах модуля A и не тратить время на изучение его подкомпонентов.

Зачем вся эта головная боль? Прежде всего, хотелось бы избежать создания "бомбы замедленного действия", той, что остается незамеченной и позже взрывается в самый неподходящий момент во время работы над проектом. Подчеркивая важность "тестирования в рамках контракта", мы пытаемся, насколько это возможно, избежать катастроф, возникающих в будущем.

Подсказка 48: Проектируйте с учетом тестирования

Когда вы проектируете модуль или даже целую программу, вы обязаны проектировать ее контракт и программу для проверки этого контракта. Проектируя программу, которая проходит тестирование и выполняет соответствующий контракт, вы можете учесть граничные условия и другие аспекты, на которые в иных случаях не обратили бы внимания. Лучше всего устранять ошибки, избежав их с самого начала. На самом деле, при создании процедуры тестирования до реализации программы вам приходится испытывать интерфейс перед тем как принять его.

Создание модульных тестов

Модульные тесты не должны оказываться где-то на периферии исходной древовидной схемы. Они должны располагаться так, чтобы с ними было удобно обращаться. В случае небольших проектов можно внедрить модульный тест в сам модуль. Для более крупных проектов можно поместить каждую из процедур тестирования в отдельный подкаталог. В любом случае необходимо помнить, что если модуль сложно отыскать, то он не будет использован.

Делая тестовую процедуру доступной, вы наделяете разработчиков, которые могут воспользоваться вашей программой, двумя бесценными ресурсами:

1. Примерами того, как использовать все функциональные возможности вашего модуля

2. Средствами построения процедур регрессионного тестирования для проверки правильности любых изменений, которые будут вноситься в программу впоследствии

Если каждый класс или модуль содержит свой собственный модульный тест, это удобно, но не всегда практично. Например, в языке Java каждый класс содержит собственную подпрограмму main. За исключением файла основного класса приложения, подпрограмма main может использоваться для запуска модульных тестов; она будет игнорироваться во время работы самого приложения. Преимущество состоит в том, что программа, отправляемая заказчику, все еще содержит тесты, которые могут использоваться для диагностики проблем, возникающих "в боевой обстановке".

При работе с языком C++ вы можете добиться того же эффекта (во время компиляции)

используя конструкцию #ifdef для выборочной компиляции программы модульного теста. Ниже представлен очень простой модульный тест на языке C++, внедренный в наш модуль и проверяющий работу функции извлечения квадратного корня с помощью подпрограммы testValue, подобной программе на языке Java, реализованной ранее:

```
#ifdef _TEST_
int main(int argc, char **argv) {
argc-; argv++; // пропускаем имя программы
 if (argc<2) {   // стандартные тесты, если аргументы не указаны
   TestValue(-4.0, 0.0);
   TestValue(0.0, 0.0);
   TestValue(2.0, 1.4142135624);
   TestValue(64.0, 8.0);
   TestValue(1.0e7, 3162.2776602);
}
else {   // в этом случае используем аргументы
  double num, expected;
  while (argc>= 2) {
  num = atof(argv[0]);
  expected = atof(argv[1]);
  testValue(num.expected);
  argc – = 2;
  argv += 2;
 }
}
return 0;
}
#endif
```

Данный модульный тест запускает минимальный набор тестов или же (при наличии аргументов) позволяет использовать внешние данные. Эта возможность могла быть задействована в сценарии запуска более полного набора тестов.

Как поступить, если корректным откликом на модульный тест является выход из программы или ее аварийное завершение? В этом случае вам необходимо выбирать запускаемый тест, указывая аргумент в командной строке. Вам также придется передать некие параметры, чтобы указать различные начальные условия для ваших тестов.

Но разработки одних модульных тестов недостаточно. Вы обязаны выполнять их и выполнять часто. Это также полезно, если класс время от времени проходит процедуру тестирования.

Применение тестовых стендов

Поскольку обычно мы пишем большое количество тестирующих программ и проводим большое количество процедур тестирования, есть смысл облегчить себе жизнь и разработать стандартный тестовый стенд для конкретного проекта. Программа main, представленная в предыдущем разделе, является весьма простым тестовым стендом, но обычно нам нужно больше функциональных возможностей.

Тестовый стенд может осуществлять универсальные операции, такие как регистрация состояния системы, анализ выходных данных на наличие ожидаемых результатов, а также выбор

и запуск конкретных процедур тестирования. Стенды могут управляться при помощи графического интерфейса, могут быть написаны на том же целевом языке, что и весь проект, или реализованы в виде сочетания сборочных файлов и сценариев на языке Perl. Простой тестовый стенд описан в ответе к упражнению 41 (см. Приложение В).

При работе с объектно-ориентированными языками и средами можно создать базовый класс, содержащий универсальные операции. Отдельные тесты могут создать подкласс и добавить специфические процедуры тестирования. Можно использовать стандартное соглашение об именовании и отражение на языке Java для формирования списка процедур тестирования в автоматическом режиме. Эта методика является прекрасным способом соблюдать принцип DRY – вам не приходится следить за списком доступных тестов. Но перед тем как взлететь и начать писать свой собственный стенд, есть смысл изучить методику xUnit Кента Бека и Эриха Гаммы [URL 22]. Они уже проделали всю сложную подготовительную работу.

Вне зависимости от выбранной вами технологии тестовый стенд обязан предоставлять следующие возможности:

• Стандартный способ определения установочной процедуры и завершения работы

• Метод выбора отдельных тестов или всех доступных тестов

• Средства анализа выходных данных на наличие ожидаемых (или неожиданных) результатов

• Стандартизированная форма отчета об обнаруженных неисправностях

Процедуры тестирования должны быть составными; другими словами, процедура тестирования может состоять из различающихся степенью детализации субтестов, которые направлены на подкомпоненты. Мы можем воспользоваться этой особенностью для тестирования отдельных компонентов или системы в целом, используя те же самые инструменты.

Специальное тестирование

Во время отладки можно прекратить создание определенных тестов "на лету". Это может быть таким же простым делом, как оператор print или ввод фрагмента программы в интерактивной оболочке отладчика или ИСР.

В конце сеанса отладки необходимо формализовать процедуру специального тестирования. Если программа прервалась один раз, скорее всего она прервется снова. Не стоит просто отбрасывать в сторону созданную процедуру тестирования; добавьте ее к существующему модульному тесту.

Например, при помощи JUnit (элемент Java из семейства xUnit) можно записать процедуру проверки извлечения квадратного корня следующим образом:

```
public class JUnitExample extends TestCase {
public JUnitExampleffinal String name) {
  super(name);
}
protected void setUpQ {
// Load up test data…
testData.addElement(new dblPair(-4.0,0.0));
testData.addElement(new dblPair(0.0,0.0));
```

```
testData.addElement(new dblPair(64.0,8.0));
testData.addElement(new dblPair(Double.MAX_VALUE, 1.3407807929942597E154));
}
public void testMySqrt() {
double num, expected, result = 0.0;
Enumeration enum = testData.elements();
while (enum.hasMoreElements()) {
dblPair p = (dblPair)enum.nextElement();
num = p.getNum();
expected = p.getExpected();
testValue(num, expected);
}
}
public static Test suite() {
TestSuite suite= new TestSuite();
suite.addTest(new JUnitExample("testMySqrt"));
return suite;
}
}
```

Пакет JUnit разработан по модульному принципу: к нему можно добавлять сколько угодно тестов, и каждый из них может, в свою очередь, являться пакетом. В дополнение к этому для управления процедурой тестирования вы можете выбрать графический или текстовый интерфейс.

Построение тестового окна

Даже самые лучшие наборы тестов скорее всего не смогут обнаружить всех «жучков»: во влажных и жарких условиях реальной эксплуатации возникает нечто, что заставляет их вылезать из деревянных изделий.

Это означает, что зачастую приходится тестировать фрагмент программного обеспечения сразу после его развертывания – с реальными данными, текущими в его жилах. В отличие от печатной платы или чипа, в программном обеспечении нет тестовых контактов, но можно по-разному взглянуть на внутреннее состояние модуля, не прибегая к помощи отладчика (в производственных условиях его применение либо неудобно, либо просто невозможно).

Одним из таких механизмов являются файлы журналов. Сообщения в журналах должны записываться в обычном последовательном формате; возможно, вы захотите провести их синтаксический анализ в автоматическом режиме дня определения времени обработки или логических путей, по которым двигалась программа. Диагностические процедуры, составленные небрежно или в несовместимом формате, вызывают тошноту – их трудно читать и непрактично анализировать.

Другим механизмом, позволяющим заглянуть внутрь выполняющейся программы, является комбинация "горячих клавиш". При нажатии такой комбинации на экране появляется окно диагностики с сообщениями о состоянии и т. д. Совсем не обязательно сообщать о такой возможности конечным пользователям, но это может быть весьма полезно для службы технического сопровождения.

Для более крупных программ, работающих на серверах, существует изящная технология, заключающаяся в том, что для слежения за ходом работы используется встроенный web-сервер.

Можно привязать web-браузер к HTTP-порту приложения

(который обычно имеет нестандартный номер типа 8080) и увидеть внутреннее состояние, журналы и даже нечто вроде панели управления отладкой. Реализация этого может показаться сложным делом, что не соответствует действительности это. Бесплатно внедряемые web-серверы с протоколом HTTP реализованы на различных современных языках программирования. Поиск можно начать с сайта [URL 58].

Культура тестирования

Все создаваемые вами программы будут протестированы – если не вами и вашей командой, то конечными пользователями, так что вы вполне можете планировать их тщательное тестирование. Небольшая предусмотрительность окажет серьезную помощь в минимизации затрат на сопровождение и снизит количество обращений в службу технического сопровождения.

Несмотря на репутацию хакеров, члены сообщества Perl являются стойкими приверженцами регрессионного и модульного тестирования. Стандартная процедура инсталляции модуля в Perl поддерживает регрессионное тестирование с помощью команды

% make test

В этом отношении сам по себе Perl не является чем-то сверхъестественным. Perl облегчает сопоставление и анализ результатов тестирования для обеспечения соответствия, но его большое преимущество состоит в том, что он является стандартом – тестирование проводится в конкретном месте и имеет предсказуемый результат. Тестирование в большей степени является вопросом культуры, а не техники, независимо от используемого вами языка.

Подсказка 49: Тестируйте ваши программы, в противном случае это сделают ваши пользователи

Другие разделы, относящиеся к данной теме:

- Мой исходный текст съел кот Мурзик
- Ортогональность
- Проектирование по контракту
- Реорганизация
- Безжалостное тестирование

Упражнения

41. Спроектируйте тестовый шаблон для интерфейса блендера для коктейлей, описанного в ответе к упражнению 17 (см. Приложение В). Напишите сценарий оболочки, который осуществит регрессионное тестирование блендера. Необходимо проверить основные функциональные возможности, ошибки и граничные условия, а также любые обязательства по контракту. Какие ограничения налагаются на изменение скорости вращения ротора блендера?

Соблюдаются ли они?

35
Злые волшебники

Никто не может отрицать – создавать приложения становится все сложнее и сложнее. В частности, пользовательские интерфейсы становятся все более утонченными. Двадцать лет назад приложение среднего масштаба обошлось бы интерфейсом "стеклянного телетайпа" (а может быть, интерфейса не было бы и вовсе). Асинхронные терминалы обеспечивали интерактивное отображение символов, а устройства ввода (наподобие вездесущей IBM 3270) позволяли набирать целую экранную страницу перед нажатием клавиши SEND. Теперь пользователи требуют графический интерфейс с контекстно-зависимой справкой, средствами типа "вырезать и вставить", "перетащить и отпустить", средством OLE, много- или однодокументным интерфейсом. Пользователям потребна интеграция с web-браузером и поддержка архитектуры с тонким клиентом.

Усложняются и сами приложения. В настоящее время большинство разработок использует многозвенную модель, возможно, с промежуточным программным обеспечением или монитором транзакций. Эти программы отличаются динамичностью, гибкостью и способностью работать во взаимодействии с приложениями, написанными сторонними фирмами.

Кажется, мы не сказали о том, что нам это было нужно на прошлой неделе – вес и сразу!

Разработчики стараются быть в форме. Если бы мы использовали те же самые инструментальные средства, которые применялись для терминалов ввода-вывода двадцатилетней давности, то ничего бы не добились.

Поэтому производители инструментальных средств и поставщики средств инфраструктуры придумали палочку выручалочку – функцию-мастера. Это замечательное средство. Вам нужно приложение с многодокументным интерфейсом и поддержкой контейнера OLE? Один щелчок мыши, ответ на пару простых вопросов – и функция-мастер автоматически сгенерирует для вас скелет программы. При выполнении данного сценария среда Microsoft Visual C++ автоматически создает программу, содержащую свыше 1200 строк. Функции-мастера хорошо справляются и с другими заданиями. Вы можете воспользоваться мастерами при создании серверных компонентов, реализации Java beans, работе с сетевыми интерфейсами – все это достаточно сложные области, где не обойтись без помощи эксперта.

Но применение функции-мастера, спроектированной неким компьютерным гуру, не делает автоматически из разработчика Джо компьютерного эксперта. Джо чувствует себя недурно – он ведь сгенерировал большое количество исходного текста и довольно элегантную на вид программу. Ему нужно лишь добавить функциональную возможность, характерную для данного приложения, и программу можно отправлять заказчику. Но покуда Джо реально не осознает сути программы, сгенерированной от его имени, он вводит самого себя в заблуждение. Он программирует в расчете на стечение обстоятельств. Функция-мастер подобна улице с односторонним движением – она лишь «вырезает» программу и затем движется далее. Если сгенерированная программа не совсем правильна (или обстоятельства изменились), а вам необходимо адаптировать ее, вы остаетесь с ней один на один.

Мы не выступаем против функций-мастеров. Напротив, их созданию в книге посвящен целый раздел "Генераторы исходных текстов". Но если вы все же используете функцию-мастера и не понимаете всей создаваемой ею программы, то не сможете управлять вашим собственным приложением. Вы не сможете сопровождать его и будете затрачивать неимоверные усилия при отладке.

Подсказка 50: Не пользуйтесь программой функции-мастера, которую не понимаете

Некоторые полагают, что это совсем уж экстремистская позиция. Они говорят, что разработчики всегда основывают свою работу на предметах, которые до конца им непонятны, – на квантовой механике в интегральных схемах, схеме прерываний в процессоре, алгоритмах, используемых при диспетчеризации процессов, программах из имеющихся библиотек и т. д. Мы согласны. И мы придерживались бы того же мнения о функциях-мастерах, если бы они представляли собой просто набор библиотечных вызовов или стандартные службы операционной системы, на которые могли положиться разработчики. Но это не так. Функции-мастера генерируют программу, которая становится неотъемлемой частью приложения, написанного разработчиком Джо. Сгенерированная программа не выносится за скобки, прячась за опрятным интерфейсом, она переплетена, строчка за строчкой, с теми функциональными возможностями, которые созданы самим Джо [40]. В конечном итоге она перестает быть программой функции-мастера и становится программой самого Джо. Никто не должен генерировать программу, не понимая ее до конца.

Другие разделы, относящиеся к данной теме:

- Ортогональность
- Генераторы исходных текстов

Вопросы для обсуждения

- Если в вашем распоряжении имеется функция-мастер построения графического интерфейса, воспользуйтесь ей для генерирования «скелета» приложения. Внимательно изучите каждую строку сгенерированной программы. Всели в ней вам понятно? Могли бы написать ее сами? Лучше написать ее самому, или же она делает то, что вам не нужно?

Глава 7
Перед тем, как начать проект

У вас никогда не возникало ощущения, что ваш проект обречен еще до его начала? Иногда так и происходит, если вначале вы не установите некоторые основополагающие правила. В противном случае вы можете объявить проект закрытым и сэкономить спонсору некоторую сумму.

В самом начале проекта вам придется определить требования. Недостаточно лишь выслушать пользователей, необходимо прочесть раздел "Карьер для добычи требований".

Житейская мудрость и управление сдерживающими факторами являются основными темами раздела "Разгадка невероятных головоломок". Неважно, какую операцию вы осуществляете – анализ, составление текста программы или тестирование, проблемы возникают все равно. Чаще они не будут настолько сложными, какими показались вначале.

Даже когда вы подумаете, что решили все проблемы, то все равно будете чувствовать неудобства, начав работать над проектом. Является ли это простым промедлением или чем-то большим? В разделе "Пока вы не готовы" предлагается совет – в какой момент благоразумно прислушаться к предостережению внутреннего голоса.

Слишком раннее начало – это проблема, но слишком долгое ожидание еще хуже. В разделе "Западня со стороны требований" обсуждаются преимущества создания спецификаций по образцу.

В разделе "Круги и стрелки" рассматриваются некоторые ловушки, в которые можно попасть при использовании формальных процессов и методологий. Неважно, насколько хорошо он продуман, и какие "лучшие случаи из практики" в нем использованы, – никакой метод не заменит мышления.

Если вы устраните эти критические аспекты до того, как проект будет запущен, вы лучше справитесь с "аналитическим параличом" и начнете выполнять реальный успешный проект.

36
Карьер для добычи требований

Совершенство достигается не тогда, когда уже нечего прибавить, но когда уже ничего нельзя отнять.

Антуан де Сент-Экзюпери, Ветер, песок и звезды, 1939

Многие книги и учебные пособия относят процедуру сбора исходных требований к начальной фазе проекта. Термин «сбор» напоминает о племени счастливых аналитиков, занимающихся собирательством камней-самородков мудрости, разбросанных по земле на фоне приглушенного звучания "Пасторальной симфонии". Этот термин напоминает о том, что все требования уже имеются в наличии, нужно лишь отыскать их, положить в корзину и весело шагать дальше.

Это не совсем так. Требования редко лежат на поверхности. Обычно они находятся глубоко под толщей предположений, неверных представлений и политики.

Подсказка 51: Не собирайте требования – выискивайте их

В поисках требований

Как распознать истинное требование, пробиваясь к нему сквозь толщу грязевых наносов? Ответ на этот вопрос и прост, и сложен одновременно.

Простой ответ состоит в том, что требование формулирует необходимость осуществления чего-либо. Грамотно составленное требование выглядит следующим образом:

• Доступ к личному делу сотрудника ограничен группой уполномоченных на то лиц.

• Температура головки блока цилиндров не должна превышать определенного критического значения, зависящего от марки двигателя.

• Редактор выделяет ключевые слова, выбор которых зависит от типа редактируемого файла.

Однако подобной четкостью могут похвастаться лишь немногие требования, что и делает их анализ весьма сложной задачей.

Первая формулировка в списке, приведенном выше, вероятно, была составлена пользователями следующим образом: "Доступ к личному делу сотрудника ограничен его руководителями и работниками отдела кадров". Является ли эта формулировка требованием? Возможно, что сегодня и является, но она воплощает бизнес-политику в абсолютной формулировке. Политика же регулярно меняется, поэтому, скорее всего, мы не захотим жестко встраивать ее в наши требования. Мы рекомендуем документировать положения политики отдельно от требований и связывать их посредством гиперссылки. Сделайте требование общей формулировкой и снабдите разработчиков информацией о политике в качестве примера того, что им придется поддерживать в реализации. В конечном счете политика конечна, как и метаданные в приложении.

Это весьма тонкое различие, но именно оно окажет серьезное воздействие на

разработчиков. Если требование сформулировано как "Доступ к личному делу сотрудника ограничен персоналом фирмы", то разработчик может прекратить составление программы проверки на том месте, где приложение обращается к этим файлам. Однако если эта формулировка звучит как "Доступ к личному делу сотрудника ограничен уполномоченными на то пользователями", то разработчик, по всей вероятности, спроектирует и реализует нечто вроде системы управления доступом. При изменении политики (а оно обязательно произойдет) потребуется лишь обновление метаданных системы. На самом деле подобный метод сбора требований приведет к созданию системы, четко структурированной для поддержки метаданных.

Различия между требованиями, политикой и реализацией могут быть весьма размытыми, если речь идет о пользовательских интерфейсах. Слова "Система должна давать возможность выбора срока предоставления ссуды" представляют собой формулировку требования. Выражение "Для выбора срока предоставления ссуды необходимо окно списка" может являться формулировкой, а может таковой и не являться. Если пользователям позарез нужно окно списка, то в этом случае речь идет о требовании. Если же вместо этого они описывают свою способность выбирать, используя окно списка лишь в качестве примера, то здесь говорится не о требовании. Врезка ниже "Когда интерфейс становится системой" описывает проект, который пошел совсем не в ту сторону, поскольку потребности пользователей в интерфейсе были проигнорированы.

Важно обнаружить основополагающую причину того, почему пользователи поступают определенным образом, а не так, как они привыкли это делать. В конечном итоге разрабатываемой программе придется решать проблемы их бизнеса, а не просто отвечать их заявленным требованиям. Документируя причины, по которым были выдвинуты требования, команда разработчиков получит бесценную информацию, необходимую для принятия ежедневных решений, связанных с реализацией.

Существует простая методика: чтобы взглянуть изнутри на требования (которые часто являются весьма недостаточными) ваших пользователей, нужно самому стать пользователем. Пишете систему для службы поддержки? Посидите пару дней на телефоне вместе с опытным сотрудником этой службы. Занимаетесь автоматизацией ручной системы управления складскими запасами? Поработайте на складе с неделю [41]. Вы получите представление о реальном использовании системы и вдобавок будете просто поражены тем, насколько просьба "Можно я посижу рядом с вами недельку и посмотрю, как вы работаете?" способствует доверию и закладывает основы ваших взаимоотношений с пользователями. Но не путайтесь у них под ногами!

> Подсказка 52: Работайте с пользователем, чтобы мыслить категориями пользователя

Добыча полезных требований важна – в это время начинают складываться связи с вашим пользовательским ядром, изучаются их ожидания и надежды на создаваемую вами систему. Более подробно это обсуждается в разделе "Большие надежды".

Документация требований

Итак, вы садитесь за стол с пользователями и начинаете выпытывать у них, что же им

нужно на самом деле. Вы столкнетесь с несколькими вероятными сценариями, описывающими, что должно делать ваше приложение. Поскольку вы остаетесь профессионалом во всем, то вам хочется опубликовать такой документ, которым все смогут пользоваться в качестве основы при обсуждении, – разработчики, конечные пользователи и спонсоры проекта. Это весьма широкая аудитория.

Ивар Джекобсон [Jac94] предложил концепцию "сценариев использования системы" для фиксирования требований. Они позволяют описывать частные случаи использования системы не с точки зрения пользовательского интерфейса, а в более абстрактном виде. К сожалению, книга И. Джекобсона несколько расплывчата в деталях, поэтому в настоящее время не существует единого мнения о том, что же считать "сценарием использования системы". Что это – формальный или неформальный термин, прозаический или структурированный документ (подобный канцелярской форме)? Каким должен быть уровень детализации (помните, что у нас весьма широкая аудитория)?

Когда интерфейс становится системой

В своей статье (журнал «Wired», январь 1999, с. 176) продюсер и музыкант Брайан Иноу описал чудо техники – новейший микшерный пульт. Этот пульт заставляет звучать все, что в принципе может звучать. И все же, вместо того, чтобы помочь музыкантам в создании лучших произведений или ускорить (или удешевить) процесс записи, он "путается под ногами", нарушая творческий процесс.

Чтобы понять, почему это происходит, необходимо взглянуть на работу инженеров студии звукозаписи. Они сводят звук интуитивно. За годы работы в студии у них вырабатывается врожденный цикл обратной связи между ушами и кончиками пальцев, управляющих плавно движущимися регуляторами, вращающимися ручками и т. д. Однако компьютерный интерфейс нового микшерного пульта не усиливал их способностей. Вместо этого он заставлял пользователей набирать текст на клавиатуре и/или щелкать мышью. Функции, обеспечиваемые этим интерфейсом, были универсальными, но они были скомпонованы неизвестными и экзотическими способами. Функции, необходимые инженерам в их работе, иногда скрывались за невразумительными названиями или же достигались за счет неестественных сочетаний базовых средств.

Эта среда характеризовалась требованием – усилить существующие навыки работы. Вместо того, чтобы раболепно дублировать то, что уже существует, нужно было обеспечить переход на новую ступень развития.

Например, хорошим подспорьем в работе инженеров звукозаписи мог бы оказаться сенсорный интерфейс, смонтированный в виде классического микшерного пульта, но при этом позволяющий программам выходить за границы, определенные фиксированными ручками и переключателями. Единственным способом завоевать рынок является обеспечение удобства во время перехода на новую ступень за счет уже известных метафор.

Этот пример также иллюстрирует нашу уверенность в том, что удачные инструменты всегда привыкают к рукам, их держащим. В данном случае речь идет о привыкании инструментов, которые создаются вами для других людей.

При рассмотрении сценариев использования системы стоит отметить их целенаправленную природу. Алистер Кокбэрн опубликовал статью, в которой описывается этот подход, а также шаблоны, используемые (строго или нестрого) при этом в качестве отправной точки ([Coc97a];

имеется Интернет-версия этой статьи [URL 46]). На рисунке 7.1. показан (в сокращении) пример подобного шаблона, на рис. 7.2 представлен пример сценария его использования.

Рис. 7.1. Шаблон сценария использования системы по А. Кокбэрну

A. ХАРАКТЕРНАЯ ИНФОРМАЦИЯ
– Цель в контексте
– Область действия
– Уровень
– Предусловия
– Условие успешного завершения
– Условие неудачного завершения
– Первичный действующий субъект
– Условие начала действия
B. ОСНОВНОЙ СЦЕНАРИЙ ПРИ УСПЕШНОМ ЗАВЕРШЕНИИ
C. РАСШИРЕНИЯ
D. ВАРИАНТЫ
E. СОПУТСТВУЮЩАЯ ИНФОРМАЦИЯ
– Приоритет
– Рабочая характеристика
– Частота
– Превосходящий прецедент использования
– Подчиненный прецедент использования
– Канал связи с первичным действующим субъектом
– Вторичные действующие субъекты
– Канал связи со вторичными действующими субъектами
F. РАСПИСАНИЕ
G. ОТКРЫТЫЕ ПРОБЛЕМЫ

Используя формальный шаблон в качестве шпаргалки, вы можете быть уверены в том, что включили всю необходимую информацию в сценарий использования системы: характеристики производительности, другие стороны-участники, приоритет, частоту использования и разнообразные ошибки и исключения, которые могут появляться неожиданно ("нефункциональные требования"). Шаблон удобен для записи комментариев пользователей, наподобие "если мы получим условие ххх, тогда нам придется сделать ууу". Шаблон может послужить в качестве готовой повестки дня при встрече с пользователями ваших программ.

Рис. 7.2. Пример сценария использования системы

ПРЕЦЕДЕНТ ИСПОЛЬЗОВАНИЯ № 5: ПРИОБРЕТЕНИЕ ТОВАРА

A. ХАРАКТЕРНАЯ ИНФОРМАЦИЯ

• Цель в контексте: Покупатель напрямую направляет коммерческий запрос в нашу фирму и ожидает отгрузки товаров и выставления счета за указанные товары.

• Область действия: Фирма

• Уровень: Итоговая информация

• Предусловия: Нам известен покупатель, его адрес, и т. д.

• Условие успешного завершения: Покупатель получает товары, мы получаем оплату.

• Условие неуспешного завершения: Мы не производим отгрузку товаров, покупатель не производит оплату.

• Первичный действующий субъект: Покупатель, любой агент (или компьютер), действующий от имени заказчика

• Условие начала действия: Получение запроса на приобретение товара.

B. ОСНОВНОЙ СЦЕНАРИЙ С УСПЕШНЫМ ЗАВЕРШЕНИЕМ

1. Покупатель обращается в фирму с запросом на приобретение товара.
2. Фирма фиксирует имя покупателя. его адрес, требуемые товары. и т. д.
3. Фирма предоставляет покупателю информацию о товарах, ценах, сроках поставки, и т. д.
4. Покупатель подтверждает заказ.
5. Фирма компонует заказ, отправляет заказ покупателю.
6. Фирма высылает покупателю счет-фактуру.
7. Покупатель оплачивает счет-фактуру.

C. **РАСШИРЕНИЯ**

3a. Один из пунктов заказа отсутствует у данной фирмы: Заказ переоформляется.

4a. Покупатель производит оплату непосредственно кредитной картой: Прием оплаты кредитной картой (прецедент использования № 44).

7a. Покупатель возвращает товар: Оформление возвращенного товара (прецедент использования № 105).

D. ВАРИАНТЫ

1. Покупатель может осуществить заказ по телефону, факсу, при помощи Интернет-формы (на странице), по другим сетям электронного обмена информацией.

7. Покупатель может оплатить заказ наличными денежным переводом, чеком, или кредитной картой.

E. СОПУТСТВУЮЩАЯ ИНФОРМАЦИЯ

• Приоритет: Высший

• Производительность: 5 минут на оформление заказа, оплата в течение 45 дней

• Частота: 200 заказов в день

• Превосходящий прецедент использования: Управление взаимоотношением с заказчиком (прецедент использования № 2).

• Подчиненные прецеденты использования: Компоновка заказа (прецедент использования № 15)

• Прием оплаты кредитной картой (прецедент использования № 44). Возврат товара покупателем (прецедент использования № 105).

• Канал общения с первичным действующим субъектом: по телефону, факсу или компьютерной сети.

• Вторичные действующие субъекты: компания – оператор платежной системы, банк, экспедиторская фирма.

F. РАСПИСАНИЕ

• Должная дата: Выпуск 1.0

G. ПРОБЛЕМЫ, ЯВЛЯЮЩИЕСЯ ОТКРЫТЫМИ

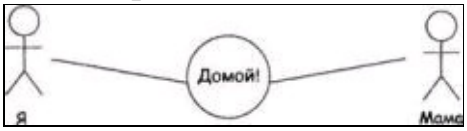
• Что происходит, если имеется лишь часть заказа?

• Что происходит, если кредитная карта похищена?

Подобного рода организация поддерживает иерархическое структурирование сценариев использования системы – вложение более подробных сценариев в сценарии более высокого уровня. Например, сценарии post debit и post credit дополняют друг друга в сценарии post transaction.

Последовательность операций может быть зафиксирована при помощи диаграмм на языке UML, а схемы концептуального представления иногда могут быть полезны для оперативного моделирования бизнес-процессов. На самом деле сценарии использования представляют собой текстовые описания с иерархией и перекрестными ссылками. Сценарии использования могут содержать гиперссылки на другие сценарии и могут вкладываться друг в друга.

Рис. 7.3. Сценарии использования, выраженные UML, понятны даже ребенку!



Кажется невероятным, что кто-нибудь может всерьез воспринимать документирование информации, используя примитивные символы, подобные изображенным на рисунке 7.3. Не будьте рабом системы обозначений: используйте любой метод общения, с помощью которого можно обмениваться требованиями с вашей аудиторией.

Чрезмерная спецификация

При генерации документов, содержащих требования, возникает серьезная опасность чрезмерной спецификации. Хорошие документы остаются абстрактными. Там, где думают о требованиях, простейшая формулировка, точно отражающая суть потребности, является наилучшей. Это не означает, что вы можете допустить неопределенность, нужно зафиксировать основополагающие семантические инварианты в качестве требований и задокументировать конкретную или же существующую на данный момент практику в качестве политики.

Требования не являются архитектурой. Требования – это не конструкция, и не пользовательский интерфейс. Это потребность.

Видеть перспективу

Вина за возникновение "проблемы 2000 года" часто возлагается на близоруких программистов, пытавшихся сэкономить несколько байтов в те дни, когда объем памяти мэйнфреймов был меньше, чем у современных пультов дистанционного управления телевизорами.

Но это не зависело от программистов и не являлось вопросом использования памяти. Если уж быть честным до конца, вина за это лежит на системных аналитиках и проектировщиках. "Проблема 2000 года" возникла по двум основным причинам: нежелание выйти за пределы существующей бизнес-практики и нарушение принципа DRY.

Двухразрядное обозначение года использовалось в деловой практике задолго до появления компьютеров. Это было обычной практикой. В то время приложения, предназначенные для обработки данных, в основном занимались автоматизацией существующих бизнес-процессов и просто повторили ошибку. Даже в том случае, когда архитектура требовала двухразрядного обозначения при вводе данных, создании отчетов и хранении данных, должна была бы появиться абстракция DATE, которая «знала» о том, что две цифры представляли собой усеченную форму реальной календарной даты.

Подсказка 53: Абстракции живут дольше, чем подробности

Требует ли от вас фраза "Видеть перспективу", чтобы вы занялись предсказанием будущего? Нет. Это означает создание формулировок типа:

> Система активно извлекает пользу из абстракции DATE. Система последовательно и универсально осуществит реализацию служб DATE наподобие форматирования, хранения данных и математических операций.

В требованиях указывается лишь то, что даты используются в принципе. Это может навести на мысль, что с датами можно производить некоторые математические действия и что даты будут храниться на различных устройствах внешней памяти. Это и есть истинные требования для модуля или класса DATE.

Еще одна мелочь…

Вина за неудачи многих проектов возлагается на увеличение области их применения – это также называется раздуванием одной их характеристик, мелким улучшательством или размыванием требований. Это аспект синдрома лягушки из раздела "Суп из камней и сварившиеся лягушки" Что можно сделать для того, чтобы требования не поглотили нас?

В литературе описаны многие метрики: количество обнаруженных и устраненных дефектов, плотность дефектов, сцепление, связывание, функциональные точки, строки программы и т. д. Эти метрики могут отслеживаться вручную или с помощью программы.

К сожалению, немногие проекты могут похвастаться активным отслеживанием требований. Это означает, что они не имеют возможности сообщать об изменении в области действия – кто затребовал средство, кто утвердил его, каково общее число утвержденных запросов и т. д.

Указание спонсорам на воздействие, оказываемое всяким новым средством на график проекта, помогает сдерживать рост количества требований. Если проект запаздывает на год по сравнению с начальными оценками, а в адрес исполнителей летят обвинения, всегда полезно иметь точную и полную картину того, как и когда происходит рост числа требований.

Легко быть втянутым в водоворот под названием "всего лишь еще одно средство", но с помощью отслеживания требований вы получите более четкое представление о том, что это "всего лишь еще одно средство" на самом деле является пятнадцатым по счету, добавленным в этом месяце.

Поддержка глоссария

Как только вы начинаете обсуждать требования, пользователи и специалисты в предметной области будут использовать определенные термины, имеющие для них специфическое значение. Например, они проводят различие между «клиентом» и «заказчиком». Было бы неуместно допустить небрежность, используя в системе то один, то другой термин.

Создайте и поддерживайте "глоссарий проекта", где будут определены все специфические термины и словарь, используемый в проекте. Все участники проекта, от конечных пользователей до специалистов службы поддержки, обязаны использовать глоссарий для обеспечения согласованности. Это подразумевает доступность глоссария для широкого круга – хороший аргумент для размещения документации на web-сайтах (об этом буквально через

минуту).

Подсказка 54: Используйте глоссарий проекта

Очень сложно создать успешный проект, в котором пользователи и разработчики обращаются к одному и тому же предмету под разными именами или, что еще хуже, обращаются к разным предметам, используя одно и тоже имя.

Прошу слова…

В разделе "Все эти сочинения" обсуждается публикация проектных документов на внутренних сайтах, обеспечивающих легкость доступа к ним со стороны всех участников. Этот способ распространения особенно полезен для документации, относящейся к требованиям.

Представляя требования в виде гипертекстового документа, мы можем обращаться к нуждам различной аудитории – дать каждому читателю, то что он хочет. Спонсоры проекта могут действовать на высоком уровне абстракции, чтобы удостовериться в том, нет что отклонений от цели бизнеса. Программисты могут использовать гиперссылки, чтобы «врубиться» в возросшие уровни детализации (даже в те, которые ссылаются на соответствующие определения или технические характеристики).

Распространение с помощью сети Интернет также позволит избежать создания толстенных отчетов под названием "Анализ требований", которые никто никогда не прочтет и которые устаревают в тот момент, когда первая капля чернил смачивает лист бумаги.

Если этот материал есть в Сети, то программисты даже могут его прочесть.

Другие разделы, относящиеся к данной теме:

- Суп из камней и сварившиеся лягушки
- Довольно приличные программы
- Круги и стрелки
- Все эти сочинения
- Большие надежды

Вопросы для обсуждения

• Можете ли вы использовать программы, которые сами пишете? Можно ли обладать хорошим чутьем на требования, будучи неспособным использовать программы самостоятельно?

• Выберите проблему (не связанную с информатикой), которую вам необходимо решить в данный момент. Сгенерируйте требования для решения, не требующего наличия компьютера.

Упражнения

42. Какие из нижеследующих примеров, по всей вероятности, являются требованиями?

Переформулируйте те, которые таковыми не являются, для придания им большей пользы (если это возможно). (Ответ см. в Приложении В.)

1. Время отклика не должно превышать 500 мс.
2. Цвет фона диалогового окна будет серым.
3. Приложение будет организовано в виде нескольких внешних процессов и внутреннего сервера.
4. Если пользователь вводит нечисловые символы в числовое поле, система будет выдавать звуковой сигнал и не примет их.
5. Приложение и данные должны умещаться в пределах 256 Кбайт.

37
Разгадка невероятных головоломок

Однажды царь Фригии Гордий завязал узел, который никто не мог развязать. Было предсказано, что тот, кто сможет развязать его, станет властелином всей Азии. И вот пришел Александр Македонский, который разрубил узел своим мечом. Несколько иная интерпретация требований и все – он стал властителем всей Азии.

Время от времени вы будете оказываться в ситуации, когда в самом разгаре проекта перед вами возникает сложнейшая головоломка: техническая проблема, с которой невозможно справиться, или фрагмент программы, составление которого оказалось намного сложнее, чем вы думали. Может быть, это выглядит просто невозможным. Но так ли это сложно на самом деле?

Рассмотрим реальные головоломки – хитроумные детальки, выполненные из дерева, металла или пластмассы, которые появляются в магазинах в дни рождественских праздников и распродаж. Задача состоит в том, что бы снять кольцо или сложить Т-образные кусочки в одну картинку, или выполнить нечто подобное.

Итак, вы пытаетесь сделать это и быстро приходите к выводу, что очевидные решения просто не срабатывают. Головоломка не может быть разгадана подобным способом. И хотя это очевидно, люди не прекращают делать одно и то же снова и снова, будучи уверенными, что это и есть нужный способ.

Конечно же, нет. Разгадка находится в совершенно другом месте. Секрет разгадки головоломки состоит в идентификации реальных (а не воображаемых) ограничений и поиске решения, исходя из этих ограничений, некоторые из которых абсолютны, а другие являются лишь предвзятыми мнениями. Абсолютные ограничения обязаны соблюдаться, какими бы неприятными и нелепыми они ни казались. С другой стороны, некоторые очевидные ограничения в реальности могут таковыми и не являться. Например, существует старый фокус, который обычно демонстрируется в баре: вы берете закупоренную бутылку шампанского и спорите, что можете пить из нее пиво. Фокус заключается в том, что вы переворачиваете бутылку донышком вверх и наливаете немного пива в углубление на донышке. Многие проблемы в программировании можно разрешить подобным оригинальным способом.

Степени свободы

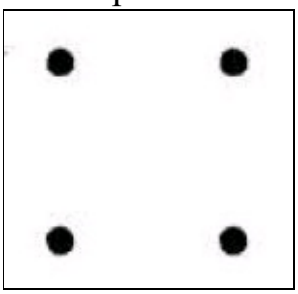
Широко известное «умное» изречение о "размышлении вне пределов ящика" поощряет распознавание ограничений, которые могут быть неприменимы в данной ситуации, и игнорирование их впоследствии. Но эта фраза не вполне точна. Если «ящик» является границей ограничений и условий, то фокус заключается в нахождении этого «ящика», который может оказаться намного больше, чем выдумаете.

Ключом к разгадке головоломки является распознавание факторов, сдерживающих вас, и степеней свободы, которые у вас есть, поскольку в них в них-то и находится разгадка. Вот почему многие головоломки столь эффективны: слишком легко вы отвергаете потенциальные разгадки.

Например, можете ли вы соединить все четыре точки (см. рисунок ниже) тремя прямыми линиями и вернуться в исходную точку, не отрывая карандаша от бумаги и не проводя одной и той же линии дважды [Но178]?

Вы обязаны бросить вызов любым предвзятым мнениям и оценить, являются ли они

реальными, раз и навсегда установленными ограничениями.



Неважно, как вы мыслите – в пределах ящика или за его пределами. Проблема заключается в нахождении ящика – распознавании реальных ограничений.

Подсказка 55: Не размышляйте вне ящика – найдите этот ящик

Столкнувшись с серьезной проблемой, представьте все возможные направления, в которых вы можете двигаться. Не отвергайте никакие варианты, какими бы бесполезными или глупыми они ни казались. Теперь просмотрите весь список и объясните, почему нельзя идти по тому или иному пути. Вы уверены в этом? Можете ли это доказать?

Рассмотрим историю с троянским конем – свежее решение непреодолимой проблемы. Как войско может попасть в укрепленный город, оставаясь незамеченным? Спорим, что вариант "через главные ворота" изначально был отвергнут как самоубийственный. Расположите ограничения по категориям и приоритетам. Столяры сначала вырезают самые крупные деревянные детали, а затем из оставшейся части – детали меньшего размера. Действуя подобным образом, мы хотим вначале идентифицировать самые жесткие ограничения и поместить оставшиеся внутрь.

Между прочим, решение головоломки с четырьмя точками представлено в Приложении В.

Есть более простой способ!

Иногда вам приходится работать над проблемой, которая оказывается намного сложнее, чем выдумали. Возможно, вы идете неправильным путем, возможно, в данный момент вы отклонились от графика выполнения проекта или уже отчаялись увидеть систему работающей, поскольку конкретную проблему "невозможно решить". В этот момент необходимо сделать шаг назад и задать себе несколько вопросов:

• Существует ли более простой способ?

• Вы пытаетесь решить главную проблему или отвлекаетесь на второстепенные технические детали?

• Почему это является проблемой?

• Что делает эту проблему столь сложной для решения?

• Стоит ли делать это именно таким образом?

• Стоит ли это делать вообще?

И во многих случаях секрет удивительным образом раскроется перед вами, как только вы попробуете ответить на один из этих вопросов. Зачастую новая интерпретация требований может унести с собой целый ворох проблем – так, как это произошло в случае с гордиевым узлом.

Все, что вам нужно, – это знание реальных ограничений, вводящих вас в заблуждение, и мудрость, позволяющая отличить одно от другого.

Вопросы для обсуждения

• Пристально взгляните на любую сложную проблему, которую вам приходится решать. Можете ли вы разрубить гордиев узел? Задайте себе ключевые вопросы, приведенные выше, особенно этот: "Стоит ли делать это именно таким образом?"

• Когда вы получили проект, которым занимаетесь в настоящее время, то прилагался ли к нему набор ограничений?

38
Чувство готовности

Тот, кто колеблется, иногда спасается

Джеймс Тэрбер, Стекло на поле

Великим лицедеям присуща одна общая черта: они знают, когда начинать, а когда подождать. Прыгун в воду стоит на трамплине, ожидая подходящего момента для прыжка. Дирижер стоит за пультом с поднятыми руками, пока не почувствует, что пора начинать.

Вы – великий артист. Вам также необходимо прислушиваться к внутреннему голосу, который шепчет «подожди». Если вы садитесь, начинаете набирать текст, а у вас в голове возникает неотступное сомнение, прислушайтесь к нему.

> Подсказка 56: Прислушайтесь к сомнениям – начинайте тогда, когда полностью готовы

Когда-то существовал стиль обучения игре в теннис под названием "внутренний теннис". Обучающийся должен был часами перекидывать мячи через сетку, не особенно заботясь о точности, а вместо этого описывая словами место, куда попал мяч относительно некой цели (часто этой целью был стул). Идея заключалась в тренировке подсознания и рефлексов, так начинающий теннисист улучшал игру, не осознавая, как и почему это происходит.

Как разработчик вы проделываете то же самое на протяжении всей вашей профессиональной карьеры. Вы испробовали разные методы и видели, какие из них работают, а какие нет. Вы накапливали опыт и мудрость. Когда сомневались или испытывали затруднения, вы учитывали это. Возможно, вы не сможете точно указать, что не так, но потерпите немного, и сомнения вероятно выкристаллизуются в нечто более основательное, на что сможете ссылаться. Разработка программ пока еще не является научной дисциплиной. Дайте инстинктам внести лепту в вашу работу.

Здравое суждение или промедление?

Каждый испытывает страх перед чистым листом бумаги. Начало нового проекта (или даже новый модуль в существующем проекте) может лишить вас спокойствия. Многие из нас предпочли бы отложить момент связывания себя обязательствами. Но вы же не можете заявить, что вы просто оттягиваете начало работы?

Создание прототипа – это методика, хорошо зарекомендовавшая себя в подобных обстоятельствах. Выберите область, которая, по вашему мнению, будет трудной, и начните создание некоего доказательства концепции. Вскоре вы можете ощутить, что тратите время понапрасну. Это признак того, что ваше изначальное сопротивление было просто желанием отложить момент связывания себя обязательствами. Откажитесь от прототипа и врубайтесь в реальную разработку.

С другой стороны, в ходе разработки прототипа наступает момент истины, когда вы внезапно осознаете, что некая базовая предпосылка была неверной. Но вам станет ясно не

только это, но и способ исправления этой предпосылки. Вы с легкостью откажетесь от прототипа и начнете проект надлежащим образом. Инстинкты не обманули, и вы сэкономили для себя и команды значительное количество усилий, которое могли потратить впустую.

Когда вы принимаете решение о создании прототипа, в целях исследования причины своего беспокойства, не забывайте, зачем это делаете. Вспоминать о том, что вы начали создавать прототип, спустя несколько недель после начала серьезного проекта – последнее дело.

Это звучит несколько цинично, но начало работ по созданию прототипа может быть более политкорректно, нежели примитивное высказывание типа "Я не настроен на начало работы" с последующим запуском игры "пасьянс".

Вопросы для обсуждения

• Обсудите синдром "страха начала работы" с вашими коллегами. Испытывают ли они тот же самый синдром? Принимают ли они его во внимание? Какие приемы они используют для его преодоления? Может ли группа преодолеть сопротивление отдельной личности, или это будет давлен и ем со стороны команды?

39
Западня со стороны требований

Пилот, совершающий посадку, не управляет самолетом до подачи команды "высота принятия решения", когда пилот, управляющий самолетом и не совершающий посадку, передает управление пилоту, не осуществляющему управление и совершающему посадку, если последний не подает команду "уход на второй круг", и в этом случае пилот, осуществляющий управление и не совершающий посадку продолжает управлять самолетом, а пилот, не управляющий самолетом и совершающий посадку, остается на связи до подачи команды «посадка» или "уход на второй круг" в зависимости от обстановки. Ввиду недавних случаев неоднозначного толкования этих правил считаем необходимым дать их более четкую формулировку.

Цитата из докладной записки авиакомпании British Airways, опубликованная в журнале "Pilot Magazine", декабрь 1996 г.

Спецификация программы – это процесс приема требований и сокращения их до точки, в которой навык программиста может взять вверх. Это акт передачи информации, объяснения и прояснения в целях устранения основных неоднозначностей. Подобно разговору с разработчиком, который будет осуществлять первоначальную реализацию, спецификация является скрижалью для будущих поколений программистов, которые будут заниматься сопровождением и усовершенствованием программы. Спецификация представляет собой также и соглашение с пользователем – это кодификация их потребностей и негласный контракт, говорящий о том, что окончательная версия системы будет соответствовать тем же требованиям.

Составление спецификации – это большая ответственность.

Проблема состоит в том, что многим проектировщикам трудно остановиться. Они полагают, что, пока каждая второстепенная деталь не будет выявлена до мельчайших подробностей, они даром получают свои деньги.

Это является ошибкой по ряду причин. Во-первых, наивно полагать, что спецификация вообще способна зафиксировать каждую подробность некой системы или предъявляемых к ней требований. В узких предметных областях существуют формальные методы, с помощью которых можно описать систему, но для объяснения смысла обозначений конечным пользователям все равно требуется проектировщик – все еще имеет место человеческий фактор. И даже в отсутствии проблем, присущих этой версии, маловероятно, что средний пользователь точно знает, что ему нужно от этого проекта. Заказчики могут сказать, что осознают суть требований и подписаться под 200-страничным документом, составленным вами, но можете быть уверены – как только они увидят систему в работе, вы будете завалены просьбами о внесении изменений.

Во вторых, существует проблема выразительности самого языка. Все методики составления диаграмм и формальные методы все еще полагаются на выражение проводимых операций средствами естественных языков [42]. А естественный язык не приспособлен для этого. Посмотрите на формулировку любого контракта: юристам приходится коверкать язык самым неестественным способом, стараясь быть точными.

Проблемный вопрос для вас. Напишите короткую инструкцию по завязыванию бантиком шнурков на ботинках. Попробуйте!

Если вы хоть чем-то похожи на нас, то скорее всего, сдадитесь, дойдя примерно до этого места: "Теперь оберните большой и указательный пальцы так, чтобы свободный конец шнурка проходил под левым шнурком во внутреннюю петлю…" Это феноменально трудное задание. И все же большинство из нас могут зашнуровать ботинки, не напрягая мозги.

Подсказка 57: Некоторые вещи лучше сделать, чем описывать

И наконец, существует "эффект смирительной рубашки" – конструкции, которая не оставляет кодировщику пространства для импровизации и отнимает усилия программирования любого рода. Кое-кто говорит, что хотел как лучше, но он неправ. Зачастую лишь на стадии написания текста некоторые варианты становятся очевидными. Во время написания программы вы можете подумать следующее: "Посмотрим вот сюда. Поскольку я написал эту подпрограмму именно таким образом, я смог добавить эту функциональную возможность практически без усилий". Или: "В спецификации говорится, что нужно сделать вот это, но я смог добиться практически того же результата, сделав по-другому, но затратил на это вдвое меньше времени". Ясно, что вы не обязаны вносить изменения, но у вас не было бы и намека на эту возможность, если бы ваши действия сдерживались конструкцией, изобилующей предписаниями.

Будучи прагматиком, вы должны стремиться рассматривать сбор требований, проектирование и реализацию как различные ипостаси одного процесса – поставки заказчику качественной системы. Не воспринимайте как изолированные друг от друга те среды, в которых происходит сбор требований, составление спецификаций и создание программ. Вместо этого постарайтесь принять «бесшовную» технологию: спецификация и реализация просто являются разными аспектами одного и того же процесса – попыткой зафиксировать и кодифицировать некое требование. Каждый из этих аспектов должен плавно переходить в другой без искусственных границ. Вы обнаружите, что в жизнеспособном процессе разработки поощряется обратная связь, идущая от реализации и тестирования к процессу составления спецификации.

Поймите нас правильно, мы не против искусственного генерирования спецификаций. Разумеется, мы признаем, что в ряде случаев необходимы невероятно подробные спецификации – в силу причин, обусловленных контрактом, из-за операционной системы, в которой вы работаете, или природы самого продукта, разработкой которого вы занимаетесь [43]. Просто осознайте, что по мере того как спецификации становятся все более подробными, их доходность начинает убывать, а то и уходит в минус. Кроме того, будьте осторожны при составлении многослойных спецификаций, нижние уровни которых не обеспечены реализацией или прототипами; слишком легко составить спецификацию того, что невозможно построить.

Чем дольше вы будете позволять спецификациям оставаться защитной оболочкой, предохраняющей разработчиков от кошмарного мира составления программ, тем сложнее будет перейти к решению задач, возникающих при составлении программ. Не окажитесь в этой спирали спецификации: в некоторой точке вам придется начать программирование! Если ваша команда будет облачена в теплые, удобные спецификации, разорвите эти оковы. Подумайте о создании прототипов или о разработке с использованием метода "стрельбы трассирующими".

Другие разделы, относящиеся к данной теме:

• Стрельба трассирующими

Вопросы для обсуждения

• Пример с завязыванием шнурка бантиком, приведенный в данной главе, является интересной иллюстрацией проблем, связанных с письменным изложением. Вы не думали о том, что лучше описывать процесс блок-схемами, а не словами? Фотографиями? С помощью некой формальной системы обозначений, взятой из топологии? При помощи модели с проволочными шнурками? Как бы вы научили ребенка завязывать шнурки?

Иногда от одного рисунка больше пользы, чем от любого количества слов. Если вы замечаете, что ваша спецификация чрезмерна, можно ли призвать на помощь рисунки или специальную систему обозначений? Насколько подробными они обязаны быть? В каких случаях лучше использовать наглядное средство, а не лекционную доску?

40
Круги и стрелки

[Фотографии] с кругами и стрелками и несколькими строками на обратной стороне, объясняющими, кто есть кто, должны были стать свидетельством против нас…

Арло Гатри, Ресторан Алисы

Начиная со структурного программирования, через бригады главного программиста, CASE-средства, разработку методом «водопада», спиральную модель, метод Джексона, диаграмму «сущность-связь», облака Буча, метод объектного моделирования, метод Objectory, метод Коуда/Йордона до современного языка UML информатика никогда не страдала от недостатка методов, стремившихся уподобить программирование инженерной дисциплине. Каждый метод имеет своих приверженцев, и каждый из них переживает период популярности. Затем ему на смену приходит следующий. Долгая жизнь была суждена возможно лишь одному из всех этих методов – структурному программированию.

И все же некоторые разработчики, дрейфуя в море тонущих проектов, продолжают цепляться за последний «пунктик», подобно тому как жертвы кораблекрушения хватаются за проплывающее бревно. Когда к ним подплывает другой обломок, то они, испытывая мучения, доплывают до него, надеясь что уж он-то будет получше. Хотя, в конце концов, качество обломка не имеет особого значения – разработчики дрейфуют все так же бесцельно.

Поймите нас правильно. Нам нравятся (некоторые) формальные методики и методы. Но мы полагаем, что слепое следование любой методике без рассмотрения ее в контексте практики разработки программ и ваших возможностей является лучшим рецептом для разочарования.

Подсказка 58: Не будьте рабом формальных методов

Формальные методы имеют ряд серьезных недостатков.

• Большинство формальных методов фиксируют требования, используя сочетание диаграмм и нескольких пояснительных фраз. На этих рисунках показано, как проектировщик понимает требования. Однако в многих случаях для конечных пользователей эти диаграммы бессмысленны, поэтому они нуждаются в их интерпретации проектировщиками. Следовательно, в реальности формальная проверка требований со стороны фактического пользователя отсутствует – все основывается на объяснениях проектировщика, как и в старомодных письменных требованиях. В этом способе фиксирования требований есть определенная польза, но мы предпочитаем, если это возможно, предоставить в распоряжение пользователя некий прототип и дать ему с ним поиграться.

• Похоже, что формальные методы поощряют специализацию. Одна группа людей работает над моделью данных, другие занимаются архитектурой, в то время как сборщики требований коллекционируют сценарии использования (или их эквивалент). Мы видели, как это приводило к плохому взаимодействию и трате усилий впустую. Кроме того, существует тенденция впадать в умонастроение типа "мы против них" – проектировщики против программистов. Мы же предпочитаем воспринимать систему, над которой работаем, целиком. Скорее всего,

невозможно будет глубоко проникнуть в суть каждого аспекта системы, но вы обязаны знать, как взаимодействуют между собой компоненты, куда помещены данные и каковы требования.

• Мы предпочитаем создавать настраиваемые динамичные системы, используя метаданные, позволяющие изменять характер приложений в ходе их выполнения. Большинство современных формальных методов сочетают модель статического объекта или данных с некоторой разновидностью механизма построения диаграммы событий или процесса. Мы пока не встречали механизма, позволяющего отображать динамизм, ожидаемый от систем. На самом деле большинство формальных методов уводят в сторону, поощряя стремление к заданию статических отношений между объектами, которые на самом деле должны быть связаны между собой динамически.

Какова отдача от методов?

В своей статье в журнале CACM [Gla99b], написанной в 1999 г., Роберт Гласе сделал обзор исследований улучшений в производительности и качестве, достигнутых благодаря семи различным технологиям разработки программ (технология 4GL, структурные методики, CASE-средства, формальные методы, методология "чистой комнаты", модели процессов и ООТ). Он сообщает, что первоначальное оживление, связанное со всеми этими методами, было преувеличено. Хотя существуют указания на то, что у некоторых методов есть преимущества, эти преимущества начинают проявляться только после существенного снижения производительности и качества, в период принятия технологии на вооружение и обучения пользователей.

Не стоит недооценивать стоимость принятия новых инструментальных средств и методов. Подготовьтесь к тому, что первые проекты с применением этих технологий будут предназначены для учебных целей.

Нужно ли использовать формальные методы?

Безусловно. Но не забывайте, что формальные методы разработки – это лишь один инструмент из вашего арсенала. Если после тщательного анализа вы почувствуете, что вам необходим формальный метод, берите его на вооружение, но помните, что несете ответственность. Никогда не становитесь рабом методологии, ведь кружки и стрелки обедняют своих хозяев. Прагматики смотрят на методологии критическим взглядом, затем берут лучшее из каждой и преобразуют их в набор практических технологий, который улучшается каждый месяц. Это является решающим моментом. Вы должны постоянно работать над усовершенствованием процессов. Никогда не делайте жесткие рамки методологии границами вашего собственного мира.

Не подавайтесь ложному авторитету метода. Люди могут ходить на собрания, принося с собой гектары бумаги с изображением диаграмм классов и сто пятьдесят сценариев использования, но вся эта макулатура – лишь их ошибочная интерпретация требований и конструкции. Старайтесь не думать о том, сколько стоит тот или иной инструмент, глядя на результаты его работы.

Подсказка 59: Дорогие инструменты не всегда создают лучшие решения

Конечно, в разработке программ есть место формальным методам. Однако, столкнувшись с проектом, философия которого заключается в изречении "диаграмма класса и есть приложение, все остальное – лишь механическое составление текста программы", знайте, что имеете дело с проектной командой, которая уцепилась за плавучее бревно и медленно гребет к берегу.

Другие разделы, относящиеся к данной теме:

• Карьер для добычи требований

Вопросы для обсуждения

• Диаграммы сценариев использования являются частью процесса UML при сборе требований (см. "Карьер для добычи требований"). Являются ли они эффективным способом взаимодействия с вашими пользователями? Если нет, то почему вы их используете?

• Как вы можете объяснить пользу, которую приносит формальный метод вашей команде? Чем вы можете ее измерить? В чем состоит улучшение? Можете ли вы провести различие между пользой от инструментального средства и возросшим опытом сотрудников вашей команды?

• Где расположена точка безубыточности при внедрении новых методов в вашей команде? Как можно оценить компромисс между пользой, приносимой в будущем, и текущими потерями в производительности в период внедрения нового инструментального средства?

• Годятся ли инструментальные средства, применяемые в крупномасштабных проектах, для малых проектов? Верно ли обратное?

Глава 8
Прагматические проекты

Поскольку вы уже работаете над проектом, нам придется отойти от вопросов, связанных с личностной философией и написанием программ, чтобы поговорить о более серьезных вещах в масштабах проекта. Мы не собираемся углубляться в специфику руководства проектами, а рассмотрим несколько критических областей, которые способны создать или разрушить любой проект.

Как только число сотрудников, работающих на проектом, превышает единицу, вам приходится устанавливать некие основные правила и делегировать части проекта соответствующим образом. В разделе "Команды прагматиков" мы покажем как это можно делать, соблюдая принципы прагматической философии.

Единственным и самый важным фактором, придающим последовательность и надежность процессам на уровне проекта, является автоматизация процедур. В разделе "Вездесущая автоматизация" мы объясним, почему это именно так, и приведем некоторые примеры из реальной жизни.

Выше говорилось о тестировании в ходе написания программ. В разделе "Безжалостное тестирование" мы переходим на следующую ступень философии и инструментов, применяемых в масштабе проекта, в особенности, если нет отдела контроля качества, находящегося у вас на побегушках.

Единственная вещь, которую разработчики не любят больше, чем тестирование, – это документация. Независимо от того, есть ли у вас технические писатели, помогающие вам, или вы пишете документацию сами, мы покажем в разделе "Все эти сочинения", как сделать эту работу менее болезненной и более продуктивной.

Успех проекта находится перед глазами наблюдателя – спонсора проекта. Восприятие успеха – это самое главное, и в разделе "Большие надежды" мы покажем вам некоторые хитрости, которые порадуют сердце любого спонсора проекта.

Последней подсказкой в этой книге является прямое следствие всех остальных. В разделе "Гордость и предубеждение" мы поощряем вас подписывать свою работу и гордиться тем, что вы делаете.

41
Команды прагматиков

В группе L Стоффел руководит шестью первоклассными программистами – это руководящая работа, которую можно приравнять к управлению бродячими котами.

Журнал "Washington Post" от 9 июня 1985 г.

Пока в книге мы рассматривали прагматические методики, которые помогают отдельной личности стать лучшим программистом. Могут ли эти методы работать в приложении к командам?

Отвечаем на это громким "да!" В личностном прагматизме есть свои преимущества, но эти преимущества преумножаются, если личность работает в команде прагматиков,

В этом разделе мы кратко рассмотрим, как прагматические методики могут применяться к целым командам. Эти замечания – лишь начало. Как только собирается команда разработчиков-прагматиков, работающих в среде, предоставляющей определенные возможности, они быстро развивают и совершенствуют свою собственную командную динамику, которая работает на них.

Рассмотрим некоторые из предыдущих разделов с точки зрения команд.

Никаких разбитых окон

Качество является прерогативой команды. Для самого прилежного разработчика, попавшего в команду, которая безразлична к работе, окажется сложным сохранять энтузиазм, необходимый для устранения проблем, требующих кропотливости. Проблемы будут только усугубляться, если команда активно уговаривает разработчика не тратить время на устранение этих проблем.

Команда в целом не должна допускать наличия разбитых окон – этих маленьких недостатков, которые никем не устраняются. Команда обязана взять на себя ответственность за качество продукта, поддерживая разработчиков, исповедующих философию "не живите с разбитыми окнами", описанную в разделе "Энтропия в программах", и поощряя ее изучение теми, кто пока не открыл ее для себя.

В некоторых методологиях коллективной работы предусмотрен менеджер по качеству – сотрудник, которому команда делегирует ответственность за качество продукта, отправляемого заказчику. Это просто смешно: качества можно достигнуть только в результате индивидуальной лепты, вносимой каждым членом команды.

Сварившиеся лягушки

Помните несчастную лягушку, сидевшую в кастрюле с водой, из разделе "Суп из камней и сварившиеся лягушки"? Она не заметила постепенного изменения в окружающей среде и в конце концов сварилась. То же самое может произойти с отдельными личностями, которые теряют бдительность. Трудно уследить за общим состоянием среды в разгаре работы над проектом.

Команда может свариться значительно быстрее, чем отдельная личность. Люди предполагают, что кто-то другой занимается неким вопросом или что руководитель команды наверняка одобрил изменение, которое просил внести пользователь. Даже самые

целеустремленные группы могут не обратить внимания на существенные изменения, происходящие с их проектами.

Боритесь с этим. Убедитесь, что каждый активно отслеживает изменения в состоянии среды. Может быть, стоит нанять "ответственного за состояние воды". Этот сотрудник должен постоянно следить за увеличением сферы покрытия, уменьшением масштабов времени, дополнительными средствами, новыми средами – всем тем, чего не было в первоначальном соглашении. Сохраняйте метрики по новым требованиям (см. раздел "Еще одна мелочь…"). Команде не нужно наотрез отказываться от изменений – просто надо знать, что они происходят. В противном случае лягушкой в горячей воде окажетесь именно вы.

Общайтесь

Очевидно, что разработчики в группе должны разговаривать друг с другом. В разделе "Общайтесь!" даны некоторые советы для облегчения подобного общения. Однако не забывайте, что сама по себе команда находится в рамках определенной организации. Команде как субъекту приходится четко взаимодействовать с остальным миром.

Для посторонних худшими проектными командами являются те, которые кажутся угрюмыми и чересчур сдержанными. Они проводят бессистемные встречи, на которых никто не хочет выступать. Их документы – сплошная путаница, среди них нет хотя бы двух похожих, и каждый разработчик пользуется своей терминологией.

Лучшие проектные команды обладают ярко выраженной индивидуальностью. Люди ожидают встреч с ними, поскольку знают, что увидят хорошо подготовленную презентацию, от которой всем станет лучше. Производимая ими документация отличается четкостью, точностью и последовательностью. В такой команде нет разноголосицы [44]. У нее даже может быть чувство юмора.

В маркетинге существует простой трюк, помогающий командам взаимодействовать как одно целое: создание брэнда. Когда вы начинаете некий проект, придумайте имя для проектной команды, в идеале – нечто из ряда вон выходящее. (В прошлом мы называли проекты в честь попугаев-киллеров, охотящихся на овец, оптических обманов и мифических городов.) Потратьте полчаса на придумывание самого идиотского логотипа и используйте его в ваших служебных записках и отчетах. В разговорах с людьми свободно упоминайте название вашей команды. Это звучит глупо, но все это придаст вашей команде некую самобытность, а миру – что-то запоминающееся, с чем можно ассоциировать вашу работу.

Не повторяйте самого себя

"В разделе "Пороки дублирования" говорилось о трудностях, связанных с устранением дублирования работы, выполняемой разными членами команды. Это дублирование ведет к тому, что усилия тратятся впустую и все выливается в кошмарные ситуации при сопровождении. Ясно, что здесь нужно четкое взаимодействие, но в ряде случаев необходимо приложить и дополнительные усилия.

Некоторые команды включают в свой состав библиотекаря проекта, который несет ответственность за координацию документации и хранение текстов исходных программ. Другие члены команды могут использовать этого сотрудника в качестве "истины в последней инстанции", когда они занимаются поиском чего-либо. Хороший библиотекарь также способен предсказать возникновение дублирования, прочитав материал, с которым они работают.

Если проект слишком велик для одного-единственного библиотекаря (или если никто не хочет брать на себя его функции), назначьте нескольких человек "фокусными точками" различных функциональных аспектов работы. Если кто-то хочет обговорить тему обработки даты, он знает, что по этому вопросу нужно обращаться к Мэри. Если же речь идет о базе данных, то следует обращаться к Фреду.

И не забудьте о значении программного обеспечения для коллективной работы и локальных телеконференциях в сети Usenet для обмена информацией и создания архивов вопросов и ответов.

Ортогональность

Традиционная организация команды основана на устаревшем методе создания программного обеспечения, известного под названием "метода водопада". Отдельным членам команды назначаются роли, основанные на их должностных обязанностях. В команде имеются бизнес-аналитики, проектировщики, программисты, тестировщики, технические писатели и т. п. [45] В этом случае существует явная иерархия – чем ближе вы допущены к конечному пользователю, тем выше ваше положение.

В стремлении довести все до крайности, некоторые объединения разработчиков диктуют строгое разграничение ответственности: тем, кто составляет программы, не разрешено общаться с теми, кто их тестирует, а им, в свою очередь, не разрешено общаться с главным архитектором и т. д. Некоторые организации еще более усложняют задачу, заставляя различные подгруппы отчитываться через отдельные цепочки управления.

Ошибочным является мнение о том, что различные действия при работе над неким проектом – анализ, проектирование, написание программы и тестирование – могут происходить изолированно друг от друга. Такого не бывает. Это различные точки зрения на одну и ту же проблему, и их искусственное разделение может вызвать целый ворох проблем. Программисты, отделенные двумя или тремя уровнями от реальных пользователей написанной ими программы, скорее всего не знают о контексте, в котором используется результат их труда. Они будут не в состоянии принять обоснованные решения.

> Подсказка 60: Организуйте команду на основе функциональности, а не должностных обязанностей

Мы одобряем разбиение команды исходя из функциональных возможностей. Разделите ваших сотрудников на небольшие группы, каждая из которых будет нести ответственность за конкретный функциональный аспект конечной версии системы. Каждая группа обладает обязательствами перед другими группами, участвующими в проекте, что определено их согласованными обязательствами. Строгий набор обязательств изменяется с каждым новым проектом, как и распределение людей по группам.

В данном случае функциональная возможность необязательно означает сценарий использования конечным потребителем программного продукта. Сюда относится и уровень доступа к базе данных, и справочная подсистема. Мы ищем сплоченные, в большой степени самостоятельные коллективы людей по тем же критериям, которые мы обязаны использовать при декомпозиции программы. Существуют признаки, предупреждающие о том, что организация команды неверна; классическим примером этого являются две подгруппы,

работающие над одним и тем же программным модулем или классом.

В чем же состоит польза от подобного функционального стиля организации? Организуя ресурсы, применяя те же методики, что и при организации программы, используя контракты (см. "Проектирование по контракту", несвязанность (см. "Несвязанность и закон Деметера") и ортогональность (раздел "Ортогональность"), мы способствуем изоляции команды в целом от влияния изменений. Если пользователь внезапно решится на замену поставщиков баз данных, то это скажется только на команде, занимающейся базами данных. Если отдел маркетинга внезапно примет решение об использовании готового средства календарного планирования, то это будет ударом только для группы разработчиков этого средства. При надлежащем исполнении подобный подход к группам может существенно снизить число пересечений в работе отдельных личностей, снизить затраты времени, повысить качество и уменьшить число дефектов. Этот подход помогает сделать команду разработчиков более сплоченной. Каждая группа знает, что только они несут ответственность за конкретную функцию.

Однако этот подход работает только при наличии ответственных разработчиков и сильного руководства. Создать пул автономных групп и позволить им разбалтываться в отсутствие руководства – это кратчайший путь к катастрофе. Проекту необходимы как минимум два руководителя – один технический, другой административный. Технический руководитель определяет философию и стиль разработки, распределяет обязанности между группами и является арбитром в неизбежных «дискуссиях» между членами команды. Он также осуществляет контроль за ситуацией в целом, стараясь найти ненужную общность задач между группами, которая снижает ортогональность общих прилагаемых усилий. Административный руководитель, или руководитель проекта, намечает ресурсы, необходимые группам, контролирует ход выполнения работ, отчитывается о проделанной работе и помогает в определении приоритетов с точки зрения потребностей бизнеса. Административный руководитель может действовать и в роли полномочного представителя команды при общении с внешним миром.

Команды, выполняющие большие проекты, нуждаются в дополнительных ресурсах: библиотекаре, который упорядочивает и хранит тексты программ и документацию, компоновщике инструментальных средств, обеспечивающем работоспособность обычных инструментальных средств и операционных сред, оперативную поддержку и т. д.

Подобная организация команды напоминает старую концепцию "бригады главного программиста", впервые описанную в 1972 г. [Bak72].

Автоматизация

Автоматизация является отличным способом обеспечить полноту и точность всего, что делает команда. Зачем компоновать текст программы вручную, если ваш редактор может делать это автоматически, пока вы набираете текст? Зачем заполнять формуляры тестирования, если процедура сборки может осуществлять тестирование автоматически?

Автоматизация является существенным компонентом любой проектной команды – настолько важным для нас, что мы посвятили ей следующий раздел, целиком. Чтобы убедиться в том, что процессы автоматизированы, назначьте одного или несколько членов группы компоновщиками инструментальных средств для конструирования и развертывания средств, автоматизирующих всю тяжелую работу. Они будут создавать файлы сборки, сценарии оболочек, шаблоны редактирования, вспомогательные программы и т. п.

Чувствуйте момент, когда нужно остановиться

Помните, что коллективы состоят из отдельных личностей. Дайте возможность каждому сотруднику проявить себя во всем блеске. Создайте структуру, достаточную для их поддержки и выполнения проекта в соответствии с требованиями. Но затем, подобно живописцу из раздела "Приемлемые программы", не поддавайтесь искушению добавить больше краски на холст.

Другие разделы, относящиеся к данной теме:

- Энтропия в программах
- Суп из камней и сварившиеся лягушки
- Приемлемые программы
- Общайтесь!
- Пороки дублирования
- Ортогональность
- Проектирование по контракту
- Несвязанность и закон Деметера
- Вездесущая автоматизация

Вопросы для обсуждения

• Оглянитесь вокруг в поисках успешных команд, работающих вне сферы разработки программного обеспечения. Каков фактор их успеха? Применяют ли они какой-либо из процессов, описанных в данном разделе?

• В следующий раз, когда вы начнете работать над проектом, постарайтесь убедить коллег в том, что проекту необходим брэнд. Дайте вашей организации время, чтобы привыкнуть к этой мысли, и затем проведите быстрый аудит, чтобы увидеть, изменило ли наличие брэнда что-нибудь как внутри команды, так и в общении с внешним миром.

• Командная алгебра: В школе мы решали задачи, наподобие этой: "Если четырем рабочим требуется 6 ч на то, чтобы выкопать канаву, то сколько времени потребуется на это восьми рабочим?" Какие факторы из реальной жизни повлияют на ответ задачи: "Если четырем программистам требуется 6 месяцев на разработку приложения, то сколько времени потребуется на это восьми программистам?" Назовите число сценариев, в которых время на разработку действительно сокращается.

42
Вездесущая автоматизация

Прогресс цивилизации состоит в расширении сферы действий, которые мы выполняем не думая.

Альфред Норт Уайтхед

На заре автомобильной эры инструкция по запуску автомобиля «Форд-Т» составляла две с лишним страницы. В современных автомобилях достаточно лишь повернуть ключ – процедура запуска является автоматической и надежной. Водитель, действующий по инструкции, может «залить» свечи зажигания, а автоматический стартер подобного не допустит.

Хотя информатика все еще напоминает автопромышленность времен выпуска модели «Форд-Т», мы не можем позволить себе в обычной работе раз за разом выполнять набор инструкций, расположенный на двух страницах. Неважно, что это – процедура сборки и выпуска готовой версии, рассмотрение текста программы или же любая повторяющаяся задача, возникающая в ходе проекта, – все это должно выполняться автоматически. Возможно, нам придется изготовить стартер и топливный инжектор "с нуля", но как только это будет сделано, с этого момента будет достаточно лишь повернуть ключ зажигания.

Мы хотим также гарантировать полноту и повторяемость при работе над проектом. Выполняемые вручную процедуры не гарантируют полноту; повторяемость также не гарантируется, особенно если аспекты конкретной процедуры открыты для интерпретации другими людьми.

Все в автоматическом режиме

Однажды мы посетили фирму-заказчик, где все разработчики использовали одну и ту же интегрированную среду разработки. Их системный администратор снабжал каждого разработчика набором инструкций по установке добавочных средств для этой среды. Эти инструкции занимали много страниц – 'щелкни мышью здесь, прокрути туда, отбуксируй это, щелкни здесь мышью два раза, повтори".

Не удивительно, что компьютер у каждого разработчика загружался по-своему. Когда разные разработчики прогоняли одну и ту же программу, в поведении приложения проявлялись малозаметные отличия. Дефекты возникали на одной машине, а на других все было нормально. При прослеживании разницы в версиях любого из компонентов обычно выявлялись неожиданные вещи.

Подсказка 61: Не используйте процедуры, выполняемые вручную

В отличие от компьютеров, люди не обладают повторяемостью в своих действиях. Мы этого от них и не ждем. Сценарий оболочки или пакетный файл выполнят те же самые инструкции, в том же порядке, раз за разом. Он может отслеживаться системой управления исходным текстом, так что возможно изучать изменения в процедуре и по прошествии времени ("но ведь она всегда работала… ").

Другим излюбленным средством автоматизации является cron (или «at» в системе Windows NT). Он позволяет планировать периодический прогон задач без участия пользователя – обычно этот прогон делается ночью. Например, представленный ниже файл crontab указывает, что команда nightly, используемая в проекте, должна запускаться каждый день в 00:05, что процедура резервного копирования backup должна запускаться в 03:15 по будням и что команда expense_eports должна выполняться в полночь первого числа каждого месяца.

```
# MIN HOUR DAY MONTH DAYOFWEEK  COMMAND
# ---------------------------------------------------
   5    0    *    *       *     /projects/Manhattan/bin/nightly
  15    3    *    *     1-5     /usr/local/bin/backup
   0    0    1    *       *     /home/accounting/expense_reports
```

С помощью cron можно планировать процедуры резервного копирования, сопровождение web-сайтов и любых других операций, которые нужно проводить без участия пользователя, т. е. автоматически.

Компилирование проекта

Компилирование проекта – это работа, которая должна быть надежной и повторяемой. Обычно проекты компилируются с помощью файлов сборки даже в интегрированной среде разработчика. В использовании файлов сборки есть ряд преимуществ. Это подготовленная по сценарию автоматическая процедура. Можно добавлять специальные программные процедуры для генерации текста программы и запускать регрессионные тесты в автоматическом режиме. Интегрированные среды имеют свои преимущества, но, пользуясь только ими, бывает трудно добиться нужного уровня автоматизации. Мы хотим осуществлять проверку, сборку, тестирование и передачу программы заказчику с помощью одной-единственной команды.

Генерирование текста программы

В разделе "Пороки дублирования" мы призываем к генерированию текстов программ для получения знания из обычных источников. Для облегчения этого процесса мы можем задействовать механизм анализа зависимости в программе make. Добавление правил в файл сборки для автоматической генерации файла из некоего другого источника не представляет особой сложности. Предположим, что есть некий файл XML, из которого необходимо сгенерировать файл Java, а результат скомпилировать.

```
.SUFFIXES: .Java .class .xml
.xml.java:
   perl convert.pl $<>$@
.java.class:
  $(JAVAC) $(JAVAC_FLAGS) $<
```

Наберем make test.class, и программа make автоматически найдет файл с именем test.XML, сформирует файл. Java, выполнив сценарий Perl, а затем скомпилирует этот файл, создав test.class.

Можно использовать подобные правила также для автоматической генерации исходного текста, файлов заголовка или документации из иной формы (см. "Генераторы исходных текстов").

Регрессионные тесты

Можно воспользоваться файлом сборки для прогона либо регрессионных тестов, либо отдельного модуля, либо подсистемы в целом. Вы легко можете протестировать весь проект целиком при помощи одной-единственной команды на вершине исходного дерева или же протестировать отдельный модуль, воспользовавшись той же командой в единственном каталоге. Более подробно регрессионное тестирование рассматривается в разделе "Безжалостное тестирование".

Рекурсивная сборка

Многие проекты устанавливают специальные рекурсивные иерархические файлы для сборки проектов и тестирования. Но не забывайте о некоторых потенциальных проблемах.

Программа make вычисляет зависимости между различными объектами, которые она должна собрать. Но она может проанализировать только зависимости, существующие в пределах одного-единственного обращения к программе make. В частности, рекурсивная программа make не обладает информацией о зависимостях, которые имеются у других обращений к программе make. Если вы будете осторожны и точны в своих действиях, то вы получите надлежащие результаты, но при этом можно проделать много лишней работы или проглядеть зависимость и не перекомпилировать ее, когда это необходимо.

Кроме того, зависимости сборки могут отличаться от зависимостей тестирования и вам могут понадобиться дополнительные иерархии.

Автоматизация процесса сборки

Сборка представляет собой процедуру, которая использует пустой каталог (и известную среду компиляции) и формирует проект с нуля, создавая то, что вы хотели бы видеть в качестве конечного результата, отправляемого заказчику, например, эталонный лазерный диск или самораспаковывающийся архив. Обычно сборка проекта включает следующие этапы:

1. Исходный текст программы извлекается из архива.

2. Проект формируется с нуля, обычно из файла сборки верхнего уровня. Каждая сборка помечается определенным номером выпуска/версии или отметкой даты.

3. Создается копия для распространения. Эта процедура может повлечь за собой фиксирование права собственности на файл и разрешения на его использование, создание всех примеров, документации, файлов README и всего того, что будет отправлено вместе с готовым продуктом именно в том формате, который требуется при передаче заказчику [46].

4. Проведите указанные тесты (процедура make test).

Для большинства проектов этот этап сборки осуществляется автоматически каждую ночь. «Ночная» сборка обычно выполняет больше полных тестов, чем отдельный сотрудник при сборке определенной части проекта. Важным моментом является то, что при полной сборке должны запускаться все тесты, имеющиеся в наличии. Вы хотите убедиться в том, что программа не прошла регрессионный тест вследствие изменений, которые были сделаны в программе сегодня. Идентифицируя проблему ближе к источнику, вы с большей вероятностью сможете отыскать и устранить существующую проблему.

Если вы не проводите регулярное тестирование, то можете обнаружить, что приложение не

работает вследствие изменения, внесенного три месяца назад. Удачи вам – в поиске этого изменения.

Окончательные сборки

Окончательные сборки, которые вы намереваетесь отправить заказчику в виде готовых продуктов, могут предъявлять требования, отличающиеся от регулярной «ночной» сборки. Окончательная сборка может требовать, чтобы библиотека исходных файлов была заблокирована или снабжена номером выпуска, чтобы флаги оптимизации и отладки были установлены по-другому и т. д. Мы предпочитаем использовать отдельный рабочий файл make (типа make final), который устанавливает все эти параметры сразу.

Помните, что если компиляция продукта отличается от компиляции предыдущей версии, то вы обязаны провести тестирование согласно этой версии заново.

Автоматические административные процедуры

Наверное, было бы недурно, если бы программисты могли реально посвящать все свое время программированию. К сожалению, это бывает очень редко. Нужно отвечать на сообщения электронной почты, выполнять бумажную работу, помещать документы в Интернет и т. д. Вы можете решиться на создание сценария оболочки, который будет делать всю грязную работу, но не забывайте запускать этот сценарий, когда необходимо.

Поскольку память – это вторая по счету вещь, которую мы теряем с возрастом [47], мы не хотим полагаться на нее слишком сильно. Мы можем запускать сценарии, которые будут выполнять для нас процедуры в автоматическом режиме, основываясь на содержимом исходного текста программы и документов. Наша цель состоит в том, чтобы поддерживать автоматическую, не требующую вмешательства пользователя последовательность операций содержательного характера.

Генерирование web-сайта

Многие команды разработчиков используют внутренний web-сайт для обмена информацией в ходе выполнения проекта, и мы полагаем, что это прекрасная идея. Но мы не хотим тратить много времени на поддержку web-сайта и не желаем, чтобы информация, содержащаяся на нем, устаревала. Информация, вводящая в заблуждение, хуже, чем отсутствие какой бы то ни было информации вообще.

Документация, извлекаемая из программы, анализа требований, проектных документов и любых чертежей, графиков или диаграмм, должна регулярно публиковаться на web-сайте. Мы предпочитаем публиковать эти документы автоматически – это является частью ночной процедуры сборки или добавочным блоком в процедуре возвращения исходного текста программы в библиотеку.

Однако если это сделано, содержание web-сайта должно генерироваться автоматически из информации, хранящейся в централизованной библиотеке, и публиковаться без вмешательства человека. На самом деле это еще одно применение принципа DRY: информация существует в одной форме – в виде исходного текста и документов в библиотеке. При просмотре с помощью web-браузера они так и выглядят – просто визуальное представление. Вам не придется

поддерживать это представление вручную.

Любая информация, сгенерированная в процессе ночной сборки, должна быть доступна на web-сайте разработчиков: результаты самой сборки (они могут быть представлены в виде краткого отчета на одной странице, содержащего предупреждения компилятора, ошибки и текущее состояние), регрессионные тесты, рабочая статистика, программные метрики, а также любые другие результаты статического анализа и т. д.

Административные процедуры утверждения

Некоторые проекты участвуют в административном документообороте, требования которого необходимо соблюдать. Например, рассмотрение проекта или текста программы должно быть спланировано и доведено до конца, документы необходимо утверждать и т. д. Можно использовать автоматизацию и – особенно это касается web-сайта – облегчить бремя, налагаемое бумажной работой.

Предположим, что вы хотели автоматизировать планирование рассмотрения и процедуру утверждения текста программы. В каждый файл с исходным текстом вы могли бы поместить специальный маркер:

/* Status: needs_review */

Простой сценарий должен пройти весь исходный текст до конца и провести поиск всех файлов, находившихся в состоянии needs_review, которое указывало на их готовность к рассмотрению. Затем вы могли бы поместить список этих файлов в виде web-страницы, автоматически послать электронную почту соответствующим адресатам или даже назначить встречу, используя программу календарного планирования.

Вы можете организовать некую форму на web-странице, чтобы рецензенты регистрировали свое утверждение или несогласие. После рассмотрения состояние может быть автоматически изменено на reviewed. Использовать или не использовать сквозной контроль текста программы всеми участниками – это зависит от вас; всю бумажную работу вы можете проделывать автоматически независимо от этого. (В своей статье в журнале CACM (апрель 1999 г.) Роберт Гласе обобщает результаты исследования, которое, похоже, указывает на то, что критическое рассмотрение текста программы отличается эффективностью, в отличие от рассмотрения в ходе собраний [Gla99a].)

Дети сапожника

Дети сапожника всегда без сапог. Зачастую те, кто занимается разработкой программ, используют наихудшие инструментальные средства для выполнения своей работы.

Но у нас имеются все исходные материалы для того, чтобы создать лучшие инструменты. У нас есть программа cron. У нас есть программа make для платформ Windows и Unix. У нас есть и Perl, а также другие языки сценариев высокого уровня для быстрой разработки заказных инструментальных средств, генераторов web-страниц, исходных тестов программ, тестовых стендов и т. д.

Пусть компьютер делает скучную земную работу – он сделает это лучше, чем мы. У нас есть задачи поважнее и потруднее.

Другие разделы, относящиеся к данной теме:

• Мой исходный текст съел кот Мурзик
• Пороки дублирования
• Сила простого текста
• Игры с оболочками
• Отладка
• Генераторы исходных текстов
• Команды прагматиков
• Безжалостное тестирование
• Все эти сочинения

Вопросы для обсуждения

• Посмотрите на свои ежедневные действия. Есть ли у вас повторяющиеся задачи? Набираете ли вы одну и туже последовательность команд раз за разом? Попробуйте написать несколько сценариев оболочки для автоматизации процесса. Всегда ли вы щелкаете мышью по определенной последовательности пиктограмм, повторяя эту операцию снова и снова? Можете ли вы создать макрокоманду, которая будет это делать за вас?

• Какая часть вашей бумажной работы, связанной с проектом, может быть автоматизирована? Учитывая большие расходы на содержание штата программистов [48], определите, какая часть проектного бюджета тратится впустую на административные процедуры. Можете ли вы обосновать временные затраты на создание автоматизированного решения, основываясь на общей экономии затрат, которая достигается при его внедрении?

43
Безжалостное тестирование

Большинство разработчиков ненавидят тестирование. Они стремятся тестировать осторожно, подсознательно ощущая, в каком месте программа может сбоить, и избегая слабых мест. Но прагматики ведут себя по-другому. Мы обладаем мотивацией к отысканию дефектов именно сейчас, чтобы нам не пришлось испытывать позор, когда кто-то другой найдет наши ошибки позже.

Поиск дефектов можно уподобить ловле рыбы с помощью сети. Мы используем мелкие, небольшие сети (модульные тесты) для ловли пескарей и большие, крупные сети (комплексные тесты) для ловли акул-убийц. Иногда рыбе удается выскользнуть, поэтому мы заделываем все найденные дыры в надежде поймать как можно больше скользких дефектов, плавающих в бассейне нашего проекта.

Подсказка 62: Тестируйте раньше. Тестируйте часто. Тестируйте автоматически

Как только у нас появляется текст программы, мы сразу хотим начать его тестирование. Крошечные пескарики имеют отвратительную привычку быстро становиться огромными акулами-людоедами, а поймать акулу намного сложнее. Но мы не хотим осуществлять все это тестирование вручную.

Многие команды разрабатывают сложные планы тестирования своих проектов. Иногда они даже их используют. Но мы обнаружили, что команды, использующие автоматизированные процедуры тестирования, имеют больше шансов на успех. Тесты, запускающиеся в ходе каждого процесса сборки, являются более эффективными по сравнению с планами тестирования, которые лежат на полке.

Чем раньше обнаружен дефект, тем дешевле обходится его устранение. "Чуть-чуть напишешь, чуть-чуть проверишь" – популярное изречение в мире Smalltalk [49], и мы можем принять эту мантру как нашу личную, создавая одновременно (или даже раньше) с написанием рабочей программы программу ее тестирования.

На самом деле удачный проект содержит больше программ тестирования, чем рабочих программ. Временные затраты на написание тестовой программы себя оправдывают. В конечном счете это оказывается намного дешевле, и вы действительно имеете возможность создания практически бездефектного продукта.

Кроме того, осознание, что вы прошли тест, дает вам большую степень уверенности в том, что этот фрагмент программы "готов".

Подсказка 63: Программа не считается написанной, пока не пройдет тестирование

Тот факт, что вы закончили работу с фрагментом программы, вовсе не означает, что можно идти к шефу или заказчику, чтобы сообщить ему о «готовности». Фрагмент не готов. Прежде всего, программа в реальности никогда не бывает готовой. И, что более важно, пока она не

пройдет все имеющиеся тесты, вы не можете утверждать, что она может использоваться кем бы то ни было.

Необходимо рассмотреть три основных аспекта тестирования в масштабе всего проекта: что тестировать, как тестировать и когда тестировать.

Что тестировать

Существует несколько видов процедур тестирования программного обеспечения, которые вам приходится выполнять:

- Модульное тестирование
- Комплексное тестирование
- Подтверждение правильности и верификация
- Тестирование в условиях нехватки ресурсов, ошибки и их исправление
- Тестирование производительности
- Тестирование удобства использования

Этот перечень ни в коей мере не является полным, и в некоторых специализированных проектах потребуются другие виды процедур тестирования. Но это дает нам хорошую отправную точку.

Модульное тестирование

Модульный тест представляет собой программу, занимающуюся тестированием некоего модуля. Эта тема освещена в разделе "Программа, которую легко тестировать". Модульное тестирование является основой для всех других видов тестирования, которые обсуждаются в данном разделе. Если части не работают по отдельности, то скорее всего они не будут хорошо работать и вместе. Все используемые модули обязаны пройти собственное модульное тестирование перед тем как продолжать работу.

Как только все соответствующие модули прошли индивидуальное тестирование, вы готовы к новому этапу. Вам придется проверить, как модули используют друг друга и взаимодействуют между собой по всей системе.

Комплексное тестирование

Комплексное тестирование показывает, что основные подсистемы, из которых состоит проект, работают и нормально взаимодействуют друг с другом. При наличии удачных и хорошо проверенных контрактов обнаружить любые проблемы интеграции не составляет особого труда. В противном случае интеграция становится благодатной почвой для размножения дефектов. Фактически в многих случаях она является единственным и самым крупным источником дефектов в системе.

В реальности комплексное тестирование является продолжением модульного тестирования, описанного выше, с той лишь разницей, что теперь вы проверяете, как целые подсистемы соблюдают свои контракты.

Подтверждение правильности и верификация

Как только в вашем распоряжении появляется рабочий пользовательский интерфейс или прототип, вам приходится отвечать на существенный вопрос: пользователи сказали вам, что они хотели бы увидеть, но то ли это на самом деле?

Отвечает ли продукт функциональным требованиям системы? Это также нуждается в тестировании. Бездефектная система, которая отвечает на неправильные вопросы, не приносит пользы. Необходимо осознавать схемы доступа конечного пользователя и их отличие от тестовых данных разработчика (в качестве примера обратите внимание на историю о рисовании кистью из раздела "Отладка")

Тестирование в условиях нехватки ресурсов, ошибки и их исправление.

Теперь вы понимаете, что система будет вести себя корректно в идеальных условиях, вам придется испытать, как она ведет себя в реальных условиях. В реальном мире ресурсы ваших программ не безграничны – они исчерпываются. Ваша программа может столкнуться со следующими ограничениями:

- Объем памяти
- Объем дискового пространства
- Мощность процессора
- Тактовая частота
- Скорость дискового обмена
- Пропускная способность сети
- Цветовая палитра
- Разрешающая способность экрана

Вы можете реально проверить нехватку дискового пространства или объема памяти, но как часто вы проверяете другие ограничения? Будет ли ваше приложение работать на экране с разрешением 640*480 и 256 цветами? Может ли оно выполняться на экране с разрешением 1600*1280 с 24-битным цветом и при этом не быть размером с почтовую марку? Завершится ли пакетное задание до момента запуска программы архивации?

Вы можете обнаружить наличие ограничений в операционной среде, таких как спецификация видеоподсистемы, и приспособиться к ним соответствующим образом. Однако не все сбои можно восстановить. Если программа обнаруживает нехватку памяти, то вы ограничены в своих действиях: вам не хватит ресурсов, чтобы завершить программу способом, отличным от аварийного завершения.

Когда система выходит из строя [50], будет ли это делаться изящно? Постарается ли она сделать лучшее, на что она способна в данной ситуации, – сохранить свое состояние и предотвратить потерю данных? Или она выдаст пользователю сообщения типа "Общая ошибка защиты" или "core dump" (отключение ядра системы)?

Тестирование производительности

Тестирование производительности, нагрузочное тестирование или тестирование в реальных условиях эксплуатации может также оказаться важным аспектом проекта.

Задайте себе вопрос, отвечает ли программа требованиям производительности в условиях реального мира – с ожидаемым числом пользователей, подключений или транзакций в единицу времени. Является ли она масштабируемой?

При работе с некоторыми приложениями вам могут понадобиться специализированные тестовая аппаратура или программное обеспечение для реалистичной имитации нагрузок.

Тестирование удобства использования

Тестирование удобства использования отличается от процедур тестирования, обсужденных выше. Оно осуществляется с реальными пользователями в реальных условиях окружающей среды.

Рассмотрим удобство использования с точки зрения человеческого фактора. Имелись ли какие-либо недоразумения в ходе анализа требований, на которые необходимо обратить внимание? Подходит ли программное обеспечение пользователю, становясь продолжением его рук? (Мы хотим, чтобы не только наши собственные инструменты были изготовлены по руке, но чтобы и те, которые мы создаем для пользователей, подходили им.)

Как и при подтверждении правильности и верификации, вам приходится осуществлять тестирование удобства использования как можно раньше, пока есть время на внесение изменений. Для крупномасштабных проектов вы можете привлечь специалистов в области человеческого фактора.

Несоответствие критериям удобства использования является дефектом такого же порядка, как деление на ноль.

Как проводить тестирование

Мы рассмотрели то, что подлежит тестированию. Теперь мы обратим внимание на то, как это делается, включая следующее:

- Регрессионное тестирование
- Тестовые данные
- Тестирование систем с графическим интерфейсом
- Тестирование самих тестов
- Исчерпывающее тестирование

Тестирование проектных решений/методологии

Можете ли вы провести тестирование проектных решений в самой программе и методологии, которую вы использовали при сборке программного обеспечения? Некоторым образом можете. Вы делаете это, анализируя метрики – измерения различных аспектов программы. Самой простой метрикой (и чаще всего, наименее интересной) является число строк кода – насколько велика сама программа?

Существует большое количество других метрик, которые вы можете использовать для исследования программы:

- Показатель цикломатической сложности Маккейба (измеряет сложность структуры решений)
- Коэффициент разветвления по входу при наследовании (количество базовых классов) и по выходу (количество производных модулей; используется в качестве родителя)
- Набор откликов (см. раздел "Несвязанность и закон Деметера")
- Отношения связывания класса (см. [URL 48])

Некоторые метрики предназначены для того, чтобы дать вам "проходной балл", тогда как другие полезны только в сравнении. Это означает, что вы вычисляете метрики для каждого модуля в системе и смотрите, как конкретный модуль относится к своим братьям. Здесь обычно используются стандартные статистические методики.

Если вы обнаруживаете модуль, чья метрика значительно отличается от всех остальных, вам необходимо задать вопрос, приемлемо ли это. Для некоторых модулей "нарушение хода кривой" может быть вполне нормально. Но для тех, у которых нет хорошего оправдания, это может свидетельствовать о потенциальных проблемах.

Регрессионное тестирование

Регрессионное тестирование сравнивает выходные данные текущего теста с результатами (или известными значениями) предыдущих. Мы можем гарантировать, что дефекты, устраненные сегодня, не нарушат ничего из того, над чем мы работали вчера. Это важное средство страховки, и оно сокращает число неприятных сюрпризов.

Все тесты, о которых мы говорили до настоящего момента, могут запускаться как регрессионные тесты с гарантией, что мы не откатываемся назад, когда разрабатываем новую программу. Мы можем запускать регрессии для тестирования производительности, контрактов, достоверности и т. д.

Тестовые данные

Где мы достаем данные для запуска всех этих тестов? Существует только два типа данных: реальные и синтезированные данные. В действительности нам необходимо использовать оба типа, поскольку их различная природа будет способствовать выявлению разных дефектов в программном обеспечении.

Реальные данные исходят из некоего реального источника. Возможно, они были получены из существующей системы, конкурирующей системы или некоего прототипа. Они представляют типичные пользовательские данные. Большие сюрпризы возникают, как только вы открываете значение термина «типичный». При этом скорее всего являются дефекты и недоразумения в анализе требований.

Синтезированные данные генерируются искусственно, возможно, с определенными статистическими ограничениями. Вам могут понадобиться синтезированные данные по одной из следующих причин:

• Вам необходимо много данных, возможно, больше, чем содержится в любом из имеющихся образцов. Вы сможете использовать реальные данные в качестве «саженца» душ генерации большего набора данных и добиться уникальности определенных полей.

• Вам необходимы данные для того, чтобы выделить определенные граничные условия. Эти данные могут быть полностью синтезированными: поля, содержащие дату 29 февраля 1999 г., огромные размеры записей или адреса с иностранными почтовыми индексами.

• Вам необходимы данные, которые демонстрируют определенные статистические свойства. Вы хотите увидеть, что случается, если сбой происходит с каждой третьей транзакцией? Вспомните алгоритм сортировки, который замедляется и ползет, когда обрабатывает предварительно отсортированные данные. Чтобы продемонстрировать эту слабость, вы можете представить данные в случайном или упорядоченном виде.

Тестирование систем с графическим интерфейсом

Тестирование систем, насыщенных графическими интерфейсами, часто требует наличия специализированных инструментальных средств. Эти средства могут основываться на простой модели захвата/воспроизведения данных или могут потребовать специально для этой цели написанных сценариев для управления графическим интерфейсом. Некоторые системы объединяют элементы обеих моделей.

Менее сложные инструментальные средства предписывают высокую степень связывания тестируемой версии программы и самого тестового сценария: если вы перемещаете диалоговое окно или уменьшаете размер экранной кнопки, процедура тестирования может не найти всего этого и оказаться неудачной. Большинство современных инструментальных средств тестирования графических интерфейсов используют ряд методик, чтобы обойти эту проблему и попытаться приспособиться к незначительным различиям в компоновке.

Однако вы не можете автоматизировать все. Энди работал над графической системой, которая позволяла пользователю создавать и отображать недетерминированные визуальные эффекты, моделирующие различные природные явления. К сожалению, в ходе тестирования нельзя просто захватить растровое изображение и сравнить с предыдущим прогоном, потому что приложение было спроектировано так, что каждый раз оно выполнялось по-разному. В подобных ситуациях у вас может не быть выбора, кроме как положиться на ручную интерпретацию результатов теста.

Одним из преимуществ, возникающих при написании несвязанной программы (см. "Несвязанность и закон Деметера") является большая доля модульного тестирования. Например, для приложений, занимающихся обработкой данных, которые имеют внешний графический интерфейс, конструкция должна быть несвязанной в достаточной степени, чтобы можно было тестировать логику приложения в отсутствии графического интерфейса. Эта идея аналогична необходимости тестировать компоненты в числе первых. Как только достоверность логики приложения подтверждается, задача по поиску дефектов, которые выявляются при наличии пользовательского интерфейса, не представляет труда (скорее всего, эти дефекты были созданы программой интерфейса пользователя).

Тестирование самих тестов

Поскольку мы не можем писать совершенные программы, то из этого следует, что мы не можем написать и совершенные программы для тестирования. Нам необходимо тестировать сами тесты.

Рассматривайте набор тестовых пакетов как сложную систему безопасности, предназначенную для подачи звукового сигнала тревоги при выявлении дефекта. Ведь нет лучшего способа проверки безопасности системы, как попытаться вломиться в нее?

После того как вы написали тест для обнаружения конкретного дефекта, вызовите этот дефект преднамеренно и удостоверьтесь, что тест его обнаружил. Это гарантия того, что тест обязательно выловит этот дефект в реальных условиях.

Подсказка 64: Используйте диверсантов для тестирования самих тестов

Если вы серьезно относитесь к тестированию, то вы должны нанять диверсанта проекта, чья роль состоит в том, чтобы воспользоваться отдельной копией исходного дерева, преднамеренно внести дефекты и проверить, что при тестировании они будут выловлены.

При написании тестов убедитесь, что сигналы тревоги раздаются тогда, когда они обязаны раздаваться.

Исчерпывающее тестирование

Вы уверены в том, что ваши тесты являются корректными и обнаруживают созданные вами дефекты. Но как вы узнаете о том, насколько исчерпывающе вы провели тестирование ядра программы?

Ответ здесь краток: «никак», вы никогда это не узнаете. Но на программном рынке есть продукты, которые могут вам помочь. Эти средства анализа степени покрытия отслеживают программу при тестировании и регистрируют, какие строки были выполнены, а какие нет. Эти средства дают общее представление о том, насколько исчерпывающей является процедура тестирования, но не стоит ожидать, что степень покрытия составит 100 %.

Даже если выполненными окажутся все строки программы, это еще не все. Важно то число состояний, в которых может находиться программа. Состояния не являются эквивалентом строк программы. Предположим, что есть функция, обрабатывающая два целых числа, каждое из которых может принимать значения от 0 до 999.

```
int test(int a, int b) {
  return a / (a + b)
}
```

Теоретически эта функция, состоящая из трех строк, имеет 1000000 логических состояний, 999999 из которых будут работать правильно, а одно – неправильно (когда a + b равно нулю). Если вам известно лишь то, что данная строка программы выполнена, то вам придется идентифицировать все возможные состояния программы. К сожалению, это очень сложная проблема. Настолько сложная, что "пока вы ее решите, солнце превратится в холодную глыбу".

> Подсказка 65: Тестируйте степень покрытия состояний, а не строк текста программы

Даже при высокой степени покрытия программы данные, используемые вами в процедуре тестирования, все еще оказывают огромное влияние, и, что более важно, порядок, в котором вы выполняете программу, может оказать самое сильное воздействие.

Когда тестировать

Многие проекты стремятся отложить процедуру тестирование на последний момент – тот, где оно будет срезано в преддверии контрольного срока [51]. Нужно начать тестирование намного раньше наступления этого срока. Как только появится какая-либо рабочая программа, ее необходимо протестировать.

Большинство процедур тестирования должно выполняться автоматически. Важно заметить,

что под термином «автоматически» мы имеем в виду и автоматическую интерпретацию результатов теста. Более подробно этот аспект рассматривается в разделе "Вездесущая автоматизация".

Мы предпочитаем проводить тестирование как можно чаще и всегда делаем это перед возвращением исходного текста в библиотеку. Некоторые системы управления исходным текстом, наподобие Aegis, могут осуществлять это автоматически. В других случаях мы просто набираем

% make test

Обычно не представляет труда запускать регрессии на всех отдельных модульных и комплексных тестах и проделывать это так часто, как это необходимо.

Но для ряда тестов частый прогон может представлять сложность. Для проведения нагрузочного тестирования могут потребоваться специальные настройки или оборудование и некоторая часть ручной работы. Эти тесты могут проводиться с меньшей частотой – возможно, еженедельно или ежемесячно. Но важно то, что они прогоняются на регулярной, запланированной основе. Если это нельзя сделать автоматически, то удостоверьтесь, что тесты включены в план вместе со всеми ресурсами, назначенными для данной задачи.

Кольцо сжимается

И наконец, мы хотели бы раскрыть единственный и самый важный принцип тестирования. Он очевиден, и практически в каждом учебнике говорится о том, что это нужно делать именно так. Но в силу некоторых причин в большинстве проектов этого все еще не делается.

Если дефект проскальзывает через сеть существующих тестов, вам необходимо добавить новый тест, чтобы поймать его в следующий раз.

Подсказка 66: Дефект должен обнаруживаться единожды

Если тестировщик обнаруживает дефект, это должно быть в первый и последний раз – обнаружение дефекта человеком. Автоматизированные тесты должны быть модифицированы для проверки наличия этого дефекта, начиная с момента его первоначального обнаружения, всякий раз, без каких-либо исключений, не обращая внимания на степень тривиальности, жалобы разработчика и его фразу "Этого больше не случится".

Потому что это снова случится. А у нас просто нет времени гоняться за дефектами, которые автоматизированные тесты не могли обнаружить. И нам придется тратить время на написание новой программы – с новыми дефектами.

Другие разделы, относящиеся к данной теме:

- Мой исходный текст съел кот Мурзик
- Отладка
- Несвязанность и закон Деметера
- Реорганизация
- Программа, которую легко тестировать
- Вездесущая автоматизация

Вопросы для обсуждения

• Можете ли вы осуществить автоматическое тестирование вашего проекта? Многие команды вынуждены дать отрицательный ответ. Почему? Слишком сложно определить приемлемые результаты? Не приведет ли к затруднениям попытка доказать спонсорам, что проект "сделан"?

Сложно ли проверить логику приложения независимо от графического интерфейса? Что можно сказать о графическом интерфейсе? О связывании?

44
Все эти сочинения

Лучше выцветшие чернила, чем отличная память.

Китайская пословица

Как правило, разработчики не размышляют над документацией слишком долго. В лучшем случае она является для них досадной необходимостью; в худшем случае она считается задачей с низким приоритетом в надежде на то, что руководство забудет о ней в конце работы над проектом.

Прагматики воспринимают документацию как неотъемлемую часть общего процесса разработки. Написание документации может быть облегчено, если вы не дублируете усилия, не теряете времени попусту и держите документацию под рукой, а если это возможно, – то в самой программе.

Эти мысли не отличаются оригинальностью и новизной; идея о брачном союзе программы и документации к ней появляется уже в работе Доналда Кнута о грамотном программировании и в утилите JavaDoc фирмы Sun. Мы хотим уменьшить противоречие между программой и документацией и вместо этого считать их двумя визуальными представлениями одной и той же модели (см. "Всего лишь визуальное представление"). На самом деле мы хотим пойти немножко дальше и применить все наши прагматические принципы к документации так, как мы применяем их к программам.

Подсказка 67: Считайте естественный язык одним из языков программирования

Существует два основных вида документации, которая готовится для проекта: внутренняя и внешняя. Внутренняя документация включает комментарии исходных текстов, документы, касающиеся проектирования и тестирования, и т. д. Внешняя документация – это то, что отправляется заказчику или публикуется для внешнего мира, например, руководство пользователя. Но вне зависимости от целевой аудитории или роли автора (разработчик он или технический писатель), вся документация является отражением программы. При наличии несоответствий программа – это то, что имеет значение.

Подсказка 68: Встраивайте документацию в проект, а не накручивайте ее сверху

Начнем с внутренней документации.

Комментарии в программе

Создать форматированные документы из комментариев и объявлений в исходном тексте довольно просто, но вначале нужно убедиться, в тексте программы действительно есть комментарии. Программа должна иметь комментарии, но слишком большое их количество

может быть так же плохо, как и малое.

В общем, комментарии должны обсуждать, почему выполняется та или иная операция, ее задачу и ее цель. Программа всегда демонстрирует, как это делается, поэтому комментирование – избыточная информация и нарушение принципа DRY.

Создание комментариев в тексте исходной программы дает отличную возможность документировать неуловимые фрагменты проекта, которые не могут документироваться где-либо еще: технические компромиссы, почему было принято то или иное решение, какие альтернативные варианты были отвергнуты и т. д.

Мы предпочитаем увидеть простой комментарий в заголовке (на уровне модуля), комментарии к существенным данным и объявлениям типов и краткие заголовки для каждого из классов и методов, описывающие, как используется именно эта функция и все ее неочевидные действия.

Имена переменных должны выбираться четко и со смыслом. Например, имя foo, не имеет смысла, так же как doit или manager, или stuff. «Венгерский» стиль именования (в котором вы кодируете информацию о типе переменной в самом ее имени) крайне нежелателен в объектно-ориентированных системах. Не забывайте, что вы (и те, кто идет за вами) будут читать текст программы много сотен раз, но писать ее будут лишь несколько раз. Не торопитесь, и напишите connectionPool вместо cp.

Имена, вводящие в заблуждение, еще хуже, чем бессмысленные. Приходилось ли вам слышать, как кто-нибудь объясняет несоответствия в унаследованном тексте программы типа: "Подпрограмма с именем getData на самом деле записывает данные на диск"? Человеческий мозг будет периодически все путать – это называется эффектом Струпа [Str35]. Вы можете поставить на себе следующий эксперимент, чтобы увидеть эффект подобных помех. Возьмите несколько цветных ручек и напишите ими названия цветов спектра. Но при этом название цвета должно быть написано только ручкой другого цвета. Вы может написать слово «синий» зеленым цветом, слово «коричневый» – красным и т. д. (В качестве альтернативы имеется набор цветов спектра, уже помещенный на наш web-сайт www.pragmaticprogrammer.com.) Как только вы написали названия цветов, постарайтесь как можно быстрее произнести вслух название цвета, которым написано каждое слово. В определенный момент вы собьетесь и станете читать названия цветов, а не сами цвета. Имена очень важны для восприятия, а имена, вводящие в заблуждение, вносят беспорядок в программу.

Вы можете документировать параметры, но задайте себе вопрос, а нужно ли это делать во всех случаях. Уровень комментариев, предлагаемый средством JavaDoc, кажется весьма приемлемым:

```
/**
* Найти пиковое (наивысшее) значение в указанном интервале дат
* @param aRange Range of dates to search for data.
* @param aThreshold Minimum value to consider.
* @return the value, or <code>null<code> if no value found
* greater than or equal to the threshold.
*/
public Sample findPeak(Date Range aRange, double aThreshold);
```

Вот перечень того, чего не должно быть в комментариях к исходному тексту программы.

• **Перечень функций, экспортируемых программой в файл.** Существуют программы, которые анализируют исходный текст. Воспользуйтесь ими, и этот перечень никогда не устареет.

• **Хронология изменений.** Для этого предназначены системы управления исходным текстом программы (см. "Управление исходным текстом"). Однако, будет полезно включить информацию о дате последнего изменения и сотруднике, который внес это изменение [52].

• **Список файлов, используемых данным файлом.** Это можно более точно определить при помощи автоматических инструментальных средств.

• **Имя файла.** Если оно должно указываться в файле, не поддерживайте его вручную. Система RCS и ей подобные могут обновлять эту информацию автоматически. При перемещении и удалении файла вам не хочется вспоминать о необходимости редактирования заголовка.

Одним из наиболее важных фрагментов информации, который обязан появиться в исходном файле, – это имя автора, не обязательно того, кто осуществлял последнюю редакцию, но имя владельца. Приложение обязательств и ответственности к исходному тексту программы творит чудеса, сохраняя людей честными (см. "Гордость и предубеждение").

Проект также может потребовать наличия определенных ссылок на авторские права или других юридических стереотипов в каждом исходном файле. Сделайте так, чтобы программа редактирования вставляла эти элементы автоматически.

При наличии обширных комментариев инструментальные средства, подобные JavaDoc [URL 7] и DOC++ [URL 21], могут извлекать и форматировать их для автоматического создания документации на уровне API. Это является одним из конкретных примеров более универсальной методики, которой мы пользуемся, – исполняемые документы.

Исполняемые документы

Предположим, что есть спецификация, которая перечисляет столбцы в таблице базы данных. Тогда мы получим отдельный набор команд SQL для создания реальной таблицы в базе данных и, по всей вероятности, некую структуру записи на языке программирования для хранения содержимого строки в таблице. Одна и та же информация повторяется три раза. Стоит изменить один из этих трех источников – и два других немедленно устареют. Это явное нарушение принципа DRY.

Для решения этой проблемы необходимо выбрать авторитетный источник информации. Это может быть спецификация, инструментальное средство для построения схем баз данных или некий третий источник. Выберем в качестве источника спецификацию. Теперь она является моделью нашего процесса. Нам необходим способ экспортирования информации, содержащейся в ней, в виде различных визуальных представлений, например, в виде схемы базы данных и записи на языке программирования высокого уровня [53].

Если документ хранится в виде простого текста вместе с командами описания документов (например, в виде HTML, LATeX. или troff), то в этом случае можно использовать такие инструментальные средства, как Perl, для извлечения схемы и ее автоматического переформатирования. Если документ хранится в двоичном формате текстового процессора, то ознакомьтесь с некоторыми вариантами действий, приведенных во врезке, данной ниже.

Теперь документ – неотъемлемая часть разработки проекта. Единственным способом изменения схемы является изменение документа. Вы гарантируете, что спецификация, схема и программа находятся в согласии. Вы сводите к минимум работу, которую необходимо выполнить для внесения каждого изменения, и можете обновлять визуальные представления изменений автоматически.

Как быть, если мой документ не хранится в формате простого текста!

К сожалению, в настоящее время все больше проектной документации составляется с помощью текстовых процессоров, сохраняющих файл на диске в некоем определенном формате. Мы говорим "к сожалению", потому что это существенно ограничивает возможности автоматической обработки документа. Но у вас в запасе имеется еще два варианта:

• **Создавайте макрокоманды.** Сейчас большинство многофункциональных текстовых процессоров содержит встроенные макроязыки. Затратив некоторое усилие, вы можете запрограммировать их таким образом, чтобы экспортировать отмеченные разделы документов в альтернативные формы, которые вам необходимы. Если программирование на таком уровне является для вас болезненной процедурой, вы всегда можете экспортировать соответствующий раздел в файл, имеющий стандартный формат простого текста, а затем воспользоваться инструментальным средством наподобие Perl для преобразования его в окончательную форму.

• **Сделайте документ подчиненным.** Вместо того, чтобы использовать документ в качестве определяющего источника, возьмите другое представление. В примере с базой данных вы хотели бы использовать схему в качестве авторитетной информации. Тогда создайте средство, которое экспортирует эту информацию в ту форму, которую документ может импортировать. Однако при этом будьте внимательны. Вы должны быть уверены, что эта информация импортируется всякий раз, когда документ выводится на печать, а не единожды, при создании этого документа.

Аналогичным образом можно генерировать документацию на уровне API из исходного текста программы, пользуясь инструментальными средствами, такими как JavaDoc и DOC++. Моделью является исходный текст программы: компилироваться может одно визуальное представление модели; другие представления предназначены для вывода на печать или просмотра на web-странице. Наша цель – работа над моделью (неважно, является ли эта модель самой программой или же каким-либо иным документом), и мы должны добиться того, чтобы все эти представления обновлялись автоматически (см. "Вездесущая автоматизация").

Внезапно документация оказывается не столь уж плохой.

Технические писатели

До этого момента мы говорили лишь о внутренней документации, той которую составляют сами программисты. Но что происходит, если в вашем проекте участвуют профессиональные технические писатели? Слишком часто программисты просто «перекидывают» материал техническим писателям и дают им возможность заработать себе на жизнь, создавая руководства пользователей, рекламные материалы и т. д.

Это является ошибкой. То, что программисты не составляют такие документы, вовсе не означает, что мы можем поступиться прагматическими принципами. Мы хотим, чтобы писатели восприняли те же основные принципы, что и прагматики, – соблюдали принципы DRY, ортогональности, а также концепцию "модель-визуальное представление", применяли автоматизацию и сценарии.

Печатать документ или ткать его на холсте?

Издаваемой бумажной документации присуща одна проблема: она может устареть, пока

будет напечатана. Документация в любой ее форме – лишь моментальный снимок.

Поэтому мы стараемся создавать всю документацию в форме, которая может быть помещена в информационную сеть, на web-страницу вместе с гиперссылками. Такое представление документации легче сохранять в обновленном виде, чем отслеживать все существующие бумажные экземпляры, уничтожать их и распространять обновленные версии. Это также является лучшим способом обращения к нуждам широкой аудитории. Однако не забывайте помещать дату или номер версии на каждой web-странице. В этом случае читатель сможет разобраться, что соответствует текущему моменту, что изменилось недавно, а что осталось неизменным.

Во многих случаях вам приходится представлять одну и ту же документацию в различных форматах: в печатном, в виде web-страницы, экранной справки, а может быть, и как слайд-шоу. Обычное решение в большой степени полагается на технологию "вырезать и вставить" на и создание нескольких независимых документов из одного оригинала. Это неудачная идея: представление документа не должно зависеть от его содержания.

Если вы пользуетесь системой описания документов, то обладаете гибкостью, чтобы реализовать столько различных выходных форматов, сколько вам нужно. Вы можете использовать

<H1>Chapter Title</H1>

для генерации новой главы в отчетной версии документа и названия нового слайда в слайд-шоу. Можно воспользоваться технологиями типа XSL и CSS [54] для генерирования множественных выходных форматов из этого описания.

Если вы используете текстовый процессор, то, по всей вероятности, будете располагать аналогичными возможностями. Если не забывали использовать стили для идентификации различных элементов документа, то, применяя различные таблицы стилей, вы можете существенным образом изменить внешний вид окончательного результата. Большинство современных текстовых процессоров позволяет конвертировать документы в форматы типа HTML для публикации на web-сайтах.

Языки разметки

Мы рекомендуем рассмотреть некоторые из современных схем описания документации для крупномасштабных проектов по документированию.

Многие авторы, пишущие на технические темы, используют в настоящее время средство DocBook для описания своих документов. DocBook представляет собой стандарт описания документов на основе SGML, который тщательно идентифицирует каждый компонент в документе. Документ можно обрабатывать процессором DSSSL для его преобразования в любое число различных форматов. Проект документации Linux использует DocBook для представления информации в форматах RTF, TeX, info, PostScript и HTML.

Пока ваше первоначальное описание достаточно насыщено, чтобы выразить все необходимые концепции (включая гиперссылки), перевод публикации в любую другой форму не составит труда и будет выполняться автоматически. Вы можете создавать интерактивную справку, руководства, описание основных свойств продукта для помещения на web-сайт и даже календарь с ежедневными советами – все из одного и того же источника, который находится в системе управления исходным текстом и собирается в ходе процедуры ночной сборки основной программы (см. "Вездесущая автоматизация").

Документация и программа – это различные визуальные представления одной и той же основополагающей модели, но лишь визуальные представления имеют право разниться. Не

позволяйте документации превращаться в гражданина второго сорта, которому запрещено участвовать в основном документообороте проекта. Обращайтесь с документацией так же бережно, как вы обращаетесь с программой, и пользователи (а также сотрудники службы сопровождения) будут петь вам осанну.

Другие разделы, относящиеся к данной теме:

- Пороки дублирования
- Ортогональность
- Преимущества простого текста
- Управление исходным текстом
- Всего лишь визуальное представление
- Программирование в расчете на стечение обстоятельств
- Карьер для добычи требований
- Вездесущая автоматизация

Вопросы для обсуждения

- Приходилось ли вам писать пояснительный комментарий для исходного текста программы, который вы только записали? Почему нет? Не было времени? Не уверены, что программа действительно работает – пробуете некую идею в виде прототипа? Впоследствии вы выбросите эту программу, не правда ли? Ведь при этом она не попадет в проект без комментариев и в экспериментальном виде, не так ли?
- Иногда неудобно документировать проектное решение исходного текста программы, поскольку это решение вам не совсем ясно – оно еще на стадии развития. Вы полагаете, что не должны тратить свои усилия впустую, описывая, как работает что-то, еще до того, как оно действительно начинает работать. Не похоже ли это на программирование в расчете на стечение обстоятельств? (См. одноименный раздел.)

45
Большие надежды

Подивитесь сему, небеса, и содрогнитесь, и ужаснитесь, говорит Господь.

Иеремия (2,12)

Компания объявляет о рекордной прибыли, а цена на ее акции падает на 20 %. Вечерняя телепрограмма финансовых новостей объясняет, что компании не удалось оправдать надежды аналитиков. Ребенок открывает дорогой рождественский подарок – и в слезы: там нет дешевой куклы, на которую он так надеялся. Проектная команда творит чудеса, реализуя феноменально сложное приложение, и лишь для того, чтобы получить ушат воды со стороны пользователей, поскольку в системе отсутствует справка.

В абстрактном смысле приложение успешно, если оно корректно реализует свои спецификации. К сожалению, это и оплачивается лишь абстрактно.

В действительности успех проекта измеряется тем, насколько он соответствует надеждам своих пользователей. Проект, не оправдавший их надежд, обречен на неудачу, неважно, насколько хорошо он соответствовал срокам. Однако, подобно родителям ребенка, ожидающего дешевую куклу, вы заходите слишком далеко и терпите неудачу.

Подсказка 69: Слегка превышайте надежды ваших пользователей

Однако выполнение этой подсказки требует некоторых усилий.

Передача надежд

Пользователи обычно приходят к вам с некоторым видением того, что они хотят. Оно может быть неполным, противоречивым или технически невыполнимым, но оно принадлежит пользователям, и, подобно ребенку в Рождество, они вкладывают в него некоторые эмоции. Вы не можете просто проигнорировать их видение.

По мере того как вы осознаете потребности пользователей, вы обнаруживаете области, в которых не сможете удовлетворить их требования, или области, где их требования слишком консервативны. Ваша роль частично заключается в передаче этого состояния. Работайте со своими пользователями так, чтобы их понимание того, что вы им поставляете, было точным. Этим необходимо заниматься на протяжении всего процесса разработки. Никогда не теряйте из виду те бизнес-задачи, которые предполагается решать с помощью вашей программы.

Некоторые консультанты называют этот процесс "управление ожиданиями" – активное управление тем, что пользователи надеются получить от их систем. Мы полагаем, что это несколько высокомерная позиция. Наша роль заключается не в том, чтобы управлять надеждами наших пользователей. Необходимо работать с ними, чтобы прийти к общему пониманию процесса разработки и конечного результата, наряду с теми ожиданиями, которые еще не выражены словами. Если команда свободно общается с внешним миром, то этот процесс практически автоматизирован; все должны понять, что ожидается и как это будет построено.

Существует ряд методик, которые могут использоваться для облегчения этого процесса. Из них наиболее важными являются "Стрельба трассирующими" и "Прототипы и памятные записки". Обе методики позволяют команде конструировать то, что может увидеть пользователь. Обе являются идеальными способами передать ваше понимание требований пользователей, обедают возможность вам и вашим пользователям практиковаться в общении друг с другом.

Небольшой довесок

Если вы работаете в тесном взаимодействии с вашими пользователями, разделяя их надежды и сообщая им о том, что вы делаете, то при завершении проекта практически не возникнет сюрпризов.

ЭТО ПЛОХО. Постарайтесь удивить ваших пользователей. Заметьте, их не надо пугать, их надо восхищать.

Дайте им немного больше, чем они ожидают. Небольшое усилие, которое потребуется, чтобы добавить в систему некое средство, ориентированное на пользователя, окупится доброжелательностью не один раз.

Прислушивайтесь к вашим пользователям в ходе работы над проектом, чтобы получить намеки на те средства, которые действительно могут их восхитить. Вот некоторые средства, добавляемые без особого труда, которые порадуют среднего пользователя:

- Всплывающая подсказка
- «Горячие» комбинации клавиш
- Краткое справочное руководство в качестве дополнения к руководству пользователя
- Расцвечивание
- Анализаторы журнала регистрации
- Автоматическая инсталляция
- Инструментальные средства проверки целостности системы
- Возможность запускать несколько версий системы в целях тренировки
- Экран-заставка, настроенный для фирмы-заказчика

Все эти вещи относительно поверхностны и особо не нагружают систему. Однако каждый из этих «довесков» говорит пользователям о том, что команда разработчиков позаботилась о создании отличной системы, предназначенной для использования в реальной жизни. Просто не забывайте о том, что работа системы не должна быть расстроена этими нововведениями.

Другие разделы, относящиеся к данной теме:

- Неплохие программы
- Стрельба трассирующими
- Прототипы и памятные записки
- Карьер для добычи требований

Вопросы для обсуждения

- Иногда самыми жесткими критиками проекта являются те, кто над ним работал. Случалось ли вам испытывать разочарование, когда ваши собственные надежды не были

оправданы тем, что вы создали? Как это могло произойти? Может быть здесь присутствует нечто большее, чем логика.

- Что говорят ваши пользователи, когда вы поставляете им готовую программу? Пропорционально ли их внимание к различным аспектам данной программы усилиям, которые вы в эти аспекты вложили? Что их восхищает?

46
Гордость и предубеждение

Вы восхищали нас довольно долго.

Джейн Остин, Гордость и предубеждение

Программисты-прагматики не уклоняются от ответственности. Вместо этого они испытывают радость, принимая вызовы и распространяя свой опыт. Если мы несем ответственность за проектное решение или фрагмент программы, мы делаем работу, которой можем гордиться.

Подсказка 70: Ставьте вашу подпись под работой

В прошлом мастеровые гордились, подписывая свою работу. Вы должны следовать их примеру.

Проектные команды все еще состоят из людей, и это вызывает сложности. В некоторых проектах идея монопольных прав на программу может вызывать трения. Люди могут начать обособляться или не выказывать желания работать над общими фундаментальными элементами. Проект может закончиться феодальной раздробленностью. У вас возникнут предубеждения относительно и вашей программы, и ваших коллег.

Этого-то мы и не хотим. Вы не должны ревниво защищать свою программу от тех, кто вторгается в ее пределы; вы должны платить людям той же монетой и относиться к программам других разработчиков с уважением. Золотое правило ("Поступай с другими так, как бы ты хотел, чтобы они поступали с тобой") и взаимоуважение среди разработчиков является важным для действия подсказки, приведенной выше.

Анонимность, особенно при работе с крупномасштабными проектами, может оказаться благодатной почвой для небрежности, ошибок, лени и неудачных программ. Слишком легко рассматривать себя лишь в качестве винтика в большой машине, высказывая неубедительные извинения в бесконечных отчетах о состоянии, а не просто создавая хорошие программы.

У программы должен быть владелец, но он не обязательно является физическим лицом. Успешный метод eXtreme Programming [URL 45], разработанный Кентом Беком, рекомендует коллективную собственность на программу (но это требует дополнительных процедур типа парного программирования в целях защиты от анонимности).

Мы хотим, чтобы вы гордились правом собственности. "Я это написал, и я стою за своей работой". Ваша подпись должна стать признанным знаком качества. Люди должны увидеть ваше имя в заголовке программы и рассчитывать на то, что она будет солидной, хорошо составленной, проверенной и документированной. Эта должна быть поистине профессиональная работа. Написанная настоящим профессионалом.

Прагматиком-программистом.

Приложение А
Информационные ресурсы

Авторы затронули в книге весьма широкий круг вопросов программирования, и этому есть объяснение: вопросы рассматривались с высоты птичьего полета. Но если бы им было уделено то внимание, которого они заслуживают, то объем книги стал бы больше на порядок.

Книга начинается с утверждения, что программисты-прагматики должны постоянно учиться. В данном приложении приводится перечень источников информации, которые могут им в этом поспособствовать.

В разделе "Профессиональные общества" приведены координаты IEEE (Institute of Electrical and Electronical Engineers – Институт инженеров по электротехнике и радиоэлектронике) и ACM (Association of Computing Machinery – Ассоциация по вычислительной технике). Мы рекомендуем программисту-прагматику вступить в ряды одного (или обоих) из этих обществ. Ниже в разделе "Собираем библиотеку" указаны периодические издания, книги и интернет-сайты, которые содержат высококачественную и ценную информацию (или просто-напросто забавны по своему содержанию).

В книге содержится много ссылок на программные ресурсы, доступные через Интернет. В разделе «Интернет-ресурсы» приводятся их адреса (URL) с кратким описанием. Но в силу природы Интернета многие из этих адресов могут устареть к моменту выхода книги в свет. Для того чтобы найти более свежие ссылки, можно воспользоваться одной из поисковых систем или же посетить наш интернет-сайт: www.pragmaticprogrammer.com и просмотреть соответствующий раздел.

И наконец, в приложении содержится библиографический список.

Профессиональные общества

У программистов существует два профессиональных объединения мирового уровня. Association of Computing Machinery – ACM [55] (Ассоциация по вычислительной технике) и Institute of Electrical and Electronical Engineers – IEEE [56] (Компьютерное общество института инженеров по электротехнике и радиоэлектронике). Всем программистам рекомендуется вступить в одно (или оба) из этих обществ. Кроме того, разработчики, проживающие вне США, могут вступить в соответствующие национальные объединения (примером может служить British Computer Society – BCS – Британское компьютерное общество).

Члены профессиональных обществ пользуются рядом преимуществ. Конференции и собрания дают возможность общения людям с общими интересами, а специальные секции и технические комитеты позволяют участвовать в выработке стандартов и рекомендаций, используемых во всем мире. Многое можно почерпнуть из их публикаций, в которых ведутся «высоколобые» дискуссии по практическим вопросам и «приземленные» разговоры по компьютерной теории.

Собираем библиотеку

Мы серьезно относимся к чтению книг. Как было замечено в разделе "Портфель знаний", хороший программист учится постоянно. Постоянное нахождение программиста в курсе выходящих книг и периодики способствует такому обучению.

Периодические издания

Люди, подобные авторам книги, хранят старые журналы и периодику до тех пор, пока вес стопки не будет достаточен для превращения нижних экземпляров в алмаз. Вот некоторые периодические издания, рекомендуемые авторами.

• **IEEE Computer.** Рассылается по подписке членам Компьютерного общества Института инженеров по электротехнике и радиоэлектронике. Этот журнал сосредоточен на практических аспектах, но не чужд и теории. Некоторые номера посвящены конкретной тематике, другие же представляют собой просто сборники интересных статей. Говоря языком связистов, журнал обладает хорошим соотношением "сигнал-шум".

• **IEEE Software.** Не менее ценная публикация Компьютерного общества Института инженеров по электротехнике и радиоэлектронике, издаваемая раз в два месяца и нацеленная на программистов-практиков.

• **Communications of the ACM.** Основной журнал, получаемый всеми членами Ассоциации по вычислительной технике; на протяжении десятилетий является отраслевым стандартом и опубликовал больше основополагающих статей, чем любой другой источник.

• **SIGPLAIM.** Журнал выпускается секцией языков программирования, входящей в ассоциацию ACM, распространяется среди членов ACM. В нем часто публикуются спецификации языков программирования, наряду с тематическими статьями для всех, кто любит "копать вглубь".

• **Dr. Dobbs Journal.** Ежемесячный журнал, распространяемый по подписке и продающийся в киосках. Этот журнал непрост для восприятия, но его статьи охватывают как практические аспекты, так и чистую теорию.

• **The Perl Journal.** Поклонникам Perl стоит подписаться на этот журнал (www.tpj.com).

• **Software Development Magazine.** Ежемесячный журнал, посвященный общим вопросам управления проектами и разработки программного обеспечения.

Еженедельные профессиональные издания

Существует несколько еженедельных газет для разработчиков и менеджеров. Эти газеты в основном представляют собой сборники фирменных пресс-релизов, перекроенных в статьи. Тем не менее, их содержание представляет ценность – оно позволяет быть в курсе событий, не отставать от объявлений о выходе новых продуктов и следить за возникновением и распадом отраслевых альянсов. Но от них не стоит ожидать наличия глубоко проработанных технических материалов.

Книги

Книги по компьютерной тематике весьма дороги, но при тщательном выборе они становятся хорошим вложением денег. Вот некоторые из них, понравившиеся авторам:

Анализ и проектирование

• **Object-Oriented Software Construction, второе издание.** Эпическое произведение Бертрана Мейера, посвященное основам объектно-ориентированного программирования, общим объемом 1300 страниц [Mey97Ь].

• **Design Patterns.** Конструктивный шаблон описывает метод решения конкретного класса задач на более высоком уровне по сравнению с фразеологизмом на языке программирования. Эта неоклассическая книга [GHJV95], написанная "бандой четырех" (группа авторов Erich Gamma, Richard Helm, Ralph Johnson и John Vlissides. – Прим. пер.), содержит описание 23 базовых конструктивных шаблонов, включая Proxy, Visitor и Singleton.

• **Analysis Patterns.** Сокровищница высококлассных архитектурных шаблонов, выбранных из множества реальных проектов и переработанных в виде книги. Относительно быстрый способ перенимания опыта моделирования, полученного в течение многих лет [Fow96].

Команды и проекты

• **The Mythical Man Month.** Классическое произведение (не так давно переработанное) Фреда Брукса, о "подводных камнях", появляющихся в ходе организации команд разработчиков [Bro95].

• **Dynamics of Software Development.** Серия коротких очерков о разработке программного обеспечения в больших командах, сосредоточенных на динамике взаимодействия членов команды между собой, а также между самой командой и внешним миром [McC95].

• **Surviving Object-Oriented Projects: A Manager's Guide.** "Вести с переднего края", сообщенные Алистером Кокбэрном, иллюстрируют многие опасности и ловушки, подстерегающие вас при управлении объектно-ориентированным проектом, особенно если это ваш первый проект. Г-н Кокбэрн дает подсказки и методики, позволяющие читателю решать наиболее общие проблемы [Coc97Ь].

Среды программирования

• **Unix.** У.Р. Стивенс написал несколько прекрасных книг, включая "Advanced Programming in the Unix Environment" и "Unix Network Programming" [Ste92, Ste98, Ste99].

• **Windows.** Книга Маршалла Брэйна "Win32 System Services" [Bra95] является кратким справочником по низкоуровневым интерфейсам прикладных программ. Труд Чарльза Петцольда "Programming Windows" [Pet98] стал своего рода Библией для разработчиков графических сред пользователя Windows.

• **C++.** Получив проект на языке C++, нужно бежать (а не идти) в книжный магазин и хватать книгу Скотта Мейера под названием "Effective C++" и к ней, возможно, книгу "More Effective C++" [Mey97a, Mey96]. При построении систем существенных размеров может понадобиться книга Джона Лакоса "Large-Scale C++ Software Design" [Lak96]. В поисках более

сложных методик стоит обратиться к труду Джима Коплина под названием "Advanced C++ Programming Styles and Idioms" [Cop92].

Кроме того, книги из серии Nutshell издательства O'Reilly (www.ora.com) дают оперативное и всестороннее освещение разнообразных тем и языков, таких как perl, yacc, sendmail, внутренней организации Windows и регулярных выражений.

Интернет

Найти в Интернете нужную информацию трудно. Вот несколько ссылок, которые проверяются авторами как минимум раз в неделю.

• **Slashdot.** Под заголовком "News for nerds. Stuff that matters" (Новости для дебилов. Материал со значением) скрывается один из крупнейших сайтов сообщества Linux. Помимо регулярно обновляемых новостей из мира Linux сайт предлагает информацию по модным технологиям и проблемам, которые волнуют разработчиков.

www.slashdot.org

• **Cetus Links.** Тысячи ссылок на тему объектно-ориентированного программирования.

www.cetus-links.org

• **WikiWikiWeb.** Центральная база данных шаблонов и обсуждение шаблонов (в Портленде, США). Не являясь особо выдающимся ресурсом, данный сайт представляет собой интересный эксперимент по коллективному редактированию идей.

www.c2.com

Интернет-ресурсы

Приведенные ниже ссылки на ресурсы, доступные в Интернете, были действительны во время работы над книгой, но (Сеть есть Сеть!) к моменту выхода ее в свет могут сильно устареть. В этом случае можно попробовать общий поиск по именам файлов или же зайти на сайт Pragmatic Programmer (www.pragmaticprogrammer.com) и проверить имеющиеся на нем ссылки.

Текстовые редокторы

Emacs и vi – не единственные межплатформенные редакторы, но они распространяются бесплатно и находят широкое применение. Пролистав любой специализированный журнал (например Dr. Dobbs), можно найти ряд коммерческих альтернатив вышеуказанным редакторам.

Редактор Emacs

Редакторы Emacs и XEmacs имеют версии как для платформы Unix, так и для Windows.

[URL 1] Редактор Emacs

www.gnu.org

Новейший «крупнокалиберный» редактор, обладающий всеми возможностями своих предшественников. Кривая обучения Emacs почти вертикальна, но вас ждет щедрое вознаграждение по мере овладения тонкостями работы. Редактор также содержит отличную программу чтения почты и новостей, адресную книгу, календарь и дневник, приключенческую игру…

[URL 2] Редактор XEmacs

www.xemacs.org

Отпочковавшись от классического редактора Emacs несколько лет назад, XEmacs отличается более корректными внутренними командами и более изящным интерфейсом.

Редактор vi

Существует как минимум 15 различных клонов редактора vi. Редактор vim переносится на большинство платформ и является хорошим выбором при работе в различных программных средах.

[URL 3]The Vim Editor

ftp://ftp.fu-berlin.de/misc/editors/vim

Цитата из документации: "Редактор vi содержит большое количество усовершенствований: многоуровневая отмена команд, многооконный интерфейс с буферами, выделение синтаксиса, редактирование в командной строке, дополнение имен файлов, экранная справка, визуальный выбор объектов, и т. д."

[URL 4] Редактор elvis

www.fh-wedel.de/elvis

Усовершенствованный клон редактора vi с поддержкой X.

[URL 5] Emacs Viper Mode

http://www.cs.sunysb.edu./"kifer/emacs.html

Viper представляет собой набор макрокоманд, которые придают редактору Emacs внешнее сходство с редактором vi. Некоторые могут поставить под сомнение разумность шага, заключающегося в расширении самого большого редактора в мире с целью эмулирования другого редактора, характерной чертой которого является компактность. Другие, напротив, считают, что он объединяет в себе лучшее из двух цивилизаций – Emacs и vi.

Компиляторы, языки программирования и средства разработки

[URL 6] The GNU C/C++ Compiler =» www.fsf.org/software/gcc/gcc.html

Один из наиболее популярных в мире компиляторов C и C++. Он также работает с Objective-C. (Во время работы над книгой проект egcs, который вначале отпочковался от gcc. находился в процессе воссоединения с последним.)

[URL 7] The Java Language from Sun

java.sun.com

Домашняя страница Java, включающая загружаемые SDK, документацию, средства обучения, новости и т. д.

[URL 8] Perl Language Home Page

www.perl.com

Эти ресурсы, относящиеся к языку Perl, предоставляются фирмой O'Reilly.

[URL 9] The Python Language

www.python.com

Объектно-ориентированный язык программирования Python является интерпретируемым и интерактивным, обладает хитроумным синтаксисом и большим количеством верных поклонников.

[URL 10) SmallEiffel

SmallEiffel.loria.fr

Компилятор GNU Eiffel работает на любом компьютере, снабженном компилятором ANSI C и средой выполнения Posix.

[URL 11] ISE Eiffel

www.eiffel.com

Фирме Interactive Software Engineering принадлежит авторство программы Design by Contract; она реализует на коммерческой основе компилятор Eiffel и сопутствующие инструментальные средства.

[URL 12] Sather

www.icsi.berkeley.edu/'sather

Sather является экспериментальным языком программирования, ведущим свое происхождение от Eiffel. Его задача – поддерживать функции высшего порядка и итерационной абстракции, а также Common Lisp, CLU или Scheme и быть таким же эффективным, как C, C++ или Fortran.

[URL 13] Visual Works

www.objectshare.com

Основной ресурс для среды VisualWorks Smalltalk. Некоммерческие версии для Windows и Linux распространяются бесплатно.

[URL 14] The Squeak Language Environment

www.squeak.cs.uiuc.edu

Squeak представляет собой бесплатно распространяемую и переносимую реализацию – Sma!ltalk-80, написанную на Squeak; может генерировать код на C для повышения производительности.

[URL 15] The TOM Programming Language

www.gerbil.org/tom

Весьма динамичный язык, ведущий свое начало от Objective-C.

[URL 16] The Beowulf Project

www.beowulf.org

Проект посвящен построению высокопроизводительных компьютеров из сетевых кластеров, состоящих из недорогих Linux-блоков.

[URL 17] iContract – Design by Contract Tool For Java

www.reliable-systems.com

Данное инструментальное средство использует формализм предварительных условий, выходных условий и инвариантов, реализовано в виде препроцессора для Java. Использует наследование, реализует кванторы существования и многое другое.

[URL 18] Nana – Logging and Assertions for C and C++

www.cs.ntu.edu.au/homepages/pjm/nana-home/index.html

Улучшенная поддержка проверки утверждений и регистрации в C и C++. Nana также обеспечивает некоторую поддержку для программы Design by Contract.

[URL 19] DDD – Data Display Debugger

www cs.tu-bs.de/softech/ddd

Бесплатный графический интерфейс конечного пользователя для отладчиков Unix.

[URL 20] John Brant's Refactoring Browser

www.cs.uiuc.edu/users/brant/Refactory

Популярный браузер, применяемый при реорганизации (язык Smalltalk).

[URL 21] DOC++ Documentation Generator

www.zib.de/Visual/software/doc++/index.html

DOC++ представляет собой систему документирования для C/C+ + и Java, которая генерирует выходные файлы в форматах LATEX и HTML для просмотра документации непосредственно из заголовка C++ или файлов класса Java.

[URL 22] xUnit – Unit Testing Framework

www.Xprogramming.com

Представляет простую, но мощную концепцию; модуль тестирования структур xUnit является полной платформой для тестирования программного обеспечения, написанного на нескольких языках.

[URL 23] The Tel Language

www.scriptics.com

Язык Tel (Tool Command Language) является языком сценариев, разработанным для упрощения процедуры встраивания в приложение.

[URL 24] Expect – Automate Interaction with Programs

www.expect.nist.gov

Расширение expect, построенное на языке Tel [URL 23], позволяет создавать сценарии взаимодействия с программами. Помимо помощи при составлении командных файлов, которые, например, осуществляют вызов файлов с удаленных серверов или расширяют возможности оболочки, expect приносит пользу и при регрессионном тестировании. Графическая версия expectk позволяет оборачивать приложения пользователя с графическим интерфейсом в

оконный интерфейс.

[URL 25] T Spaces

www.almaden.ibm.com.cs/TSpaces

Цитата с web-страницы: "T Spaces представляет собой сетевой коммуникационный буфер с функциональными возможностями баз данных. Он осуществляет связь между приложениями и устройствами в сети с гетерогенными компьютерами и операционными системами. T Spaces обеспечивает следующие средства: коллективной связи, работы с базами данных, переноса файлов (основанные на URL) и оповещения о событиях".

[URL 26] javaCC – Java Compiler-Compiler

www.suntest.com

Генератор грамматического разбора, связанный с языком Java.

[URL 27] The bison Parser Generator

www.gnu.org/software/bison/bison.html

Генератор bison получает на входе описание грамматики и генерирует из него исходный текст соответствующей программы грамматического разбора на языке C.

[URL 28] SWIG – Simplified Wrapper and Interface Generator

www.swig.org

SWIG представляет собой инструментальное средство разработки, стыкующее между собой программы, написанные на языках C, C++ и Objective-C, с языками высокого уровня, такими как Perl, Python, Tcl/Tk, а также Java, Eiffel и Guile.

[URL 29] The Object Management Group, Inc.

www.omg.org

Фирма Object Management Group, Inc. является «распорядителем» различных спецификаций для разработки распределенных объектно-базирующихся систем. К числу работ этой фирмы относятся CORBA (обобщенная архитектура брокера объектных запросов) и ПОР (протокол передачи сообщений между сетевыми объектами через Интернет). Сочетание этих спецификаций дает возможность объектам связываться друг с другом, даже если они написаны на разных языках и выполняются на компьютерах различных типов.

Инструментальные средство UNIX, работающие в среде DOS

[URL 30] The UWIN Development Tools

www.gtlinc.com/Products/Uwin/uwin.html

Фирма Global Technologies, Inc., Old Bridge, NJ

Пакет UWIN предоставляет библиотеки динамической компоновки (DLL) Windows, которые эмулируют большую часть библиотечного интерфейса уровня Unix C. Используя данный интерфейс, фирма Global Technologies, Inc. перенесла большое число инструментальных средств из командной строки Unix в систему Windows. См. также [URL 31].

[URL 31] The Cygnus Cygwin Tools

www.sourceware.cygnus.com/cygwin/

Фирма Cygnus Solutions, Sunnyvale, CA

Пакет Cygnus также эмулирует интерфейс библиотеки Unix C и предоставляет большой набор инструментальных средств, работающих в режиме командной строки Unix, при работе в операционной системе Windows.

[URL 32] Perl Power Tools

www.perl.com/language/ppt/

Данный проект посвящен повторной реализации классического набора команд Unix на языке Perl, что дает возможность их использования при работе со всеми платформами, поддерживающими Perl (их довольно много).

Средствауправленияисходнымтекстомпрограмм

[URL 33] RCS – Revision Control System

www.cyclic.com

Система управления исходным текстом программ GNU для Unix и Windows NT.

[URL 34] CVS – Concurrent Version System

www.cyclic.com

Система управления исходным текстом программ для Unix и Windows NT, распространяемая бесплатно. Расширяет возможности Revision Control System, поддерживая модель «клиент-сервер» и параллельный доступ к файлам.

[URL 35] Aegis Transaction-Based Configuration Management

www.canb.auug.org.au/"mil!erp/aegis.html

Инструментальное средство управления версиями (ориентированное на процесс), которое применяет к ним существующие стандарты проекта (например, проверку прохождения тестов для возвращаемых программных кодов).

[URL 36] ClearCase

www.rational.com

Управление версиями программы, рабочей областью и полной сборкой программы, управление процессом.

[URL 37] MKS Source Integrity

www.mks.com

Управление версиями и конфигурацией. Некоторые версии включают средства, позволяющие удаленным пользователям-разработчикам одновременно работать над одними и теми же файлами.

[URL 38] PVCS Configuration Management

www.merant.com

Система управления исходным текстом программ, очень популярная при работе в Windows.

[URL 39] Visual SourceSafe

www.microsoft.com

Система управления версиями, интегрируемая с инструментами визуальной разработки фирмы Microsoft.

[URL 40] Perforce

www.perforce.com

Менеджер конфигурирования программного обеспечения "клиент-сервер".

Прочие инструментальные средства

[URL 41] Winzip – Archive Utility for Windows

www.winzip.com

Фирма Nico Mak Computing, Inc., Mansfield, CT

Утилита архивирования файлов, работающая в среде Windows. Поддерживает форматы zip и tar.

[URL 42] The Z Shell

www.sunsite.auc.dk/zsh

Оболочка, предназначенная для интерактивной работы и содержащая мощный язык сценариев. В оболочку zsh было включено много полезных средств из оболочек bash, ksh и tcsh и добавлен ряд оригинальных элементов.

[URL 43] A Free SMB Client for Unix Systems
www.samba.anu.edu.au/pub/samba/

Дает возможность совместного использования файлов и других ресурсов из операционных систем Unix и Windows. Samba включает в себя:

• Сервер SMB, предоставляющий средства для работы с файлами и для печати (схожие со средствами, предоставляемыми Windows NT и LAN Manager) клиентам SMB, в роли которых могут выступать Windows 95, Warp Server, smbfs и др.

• Сервер имен Netbios, обеспечивающий, кроме всего прочего, поддержку функций браузера. По желанию пользователя Samba может быть главным браузером в локальной сети.

• Клиент SMB (схожий с клиентом ftp), позволяющий получать доступ к ресурсам ПК(дискам и принтерам) из Unix, Netware и других операционных систем.

Статьи и публикации

[URL 44] The comp.object FAQ
www.cyberdyne-object-sys.com/oofaq2

Солидный, четко организованный список часто задаваемых вопросов по группе новостей comp.object.

[URL 45] extreme Programming
www.Xprogramming.com

Цитата с интернет-сайта: "При создании команды, способной быстро создать исключительно надежное, эффективное, четко структурированное программное обеспечение, в XP используется весьма легковесное сочетание методик. Многие из методик XP создавались и опробовались в части проекта C3 фирмы «Крайслер», представляющего собой весьма успешную систему расчета заработной платы, написанную на языке Smalltalk".

[URL 46] Alistair Cockburn's Home Page
www.members.aol.com/acockburn

Стоит посмотреть раздел "Structuring Use Cases with Goals" и так называемые шаблоны сценариев использования.

[URL 47] Martin Fowler's Home Page
www.ourworld.compuseimcom/homepages/martinjowler

Мартин Фаулер является автором книги "Analysis Patterns" и соавтором книг "UML Distilled" и "Refactoring: Improvingthe Design of Existing Code". На домашней странице автора обсуждаются его книги и работа с UML.

[URL 48] Robert C. Martin's Home Page
www.objectmentor.com/home

Неплохое собрание статей ознакомительного плана по объектно-ориентированным методам, включая анализ зависимости и метрики.

[URL 49] Aspect-Oriented Programming
www.parc.xerox.com/csl/projects/aop/

Описывается методика придания функциональности программному коду с позиций ортогональности и описательности.

[URL 50] JavaSpaces Specifications
www.java.sun.com/products/javaspaces

Linda-подобная система для Java, поддерживающая распределенное сохранение состояния объекта и распределенные алгоритмы.

[URL 51] Netscape Source Code
www.mozilla.org
Исходный текст браузера Netscape.
[URL 52] The Jargon File
www.jargon.org
Eric S. Raymond
Определения дня многих общих (и не очень общих) терминов, применяемых в компьютерной индустрии, сдобренные хорошей дозой соответствующего фольклора.
[URL 53] Eric S. Raymond's Papers
www.tuxedo.org/~esr
Статьи Эрика Раймона "The Cathedral and the Bazaar" и "HomesteadingNoosphere", в которых описаны психосоциальные основы и смысл движения Open Source.
[URL 54] The K Desktop Environment
www.kde.org
Цитата с web-страницы: "KDE представляет собой мощную графическую настольную среду для рабочих станций Unix. KDE является интернет-проектом, открытым в полном смысле этого слова".
[URL 55] The GNU Image Manipulation Program
www.gimp.org
Gimp является бесплатной программой по созданию, композиции и ретушированию изображений.
[URL 56] The Demeter Project
www.ccs.neu.edu/research/demeter
Исследовательский проект, призванный упростить поддержку и развитие программного обеспечения с помощью адаптивного программирования.

Другие источники

[URL 57] The GNU Project
www.gnu.org
Фонд Free Software Foundation, Boston, MA
Фонд "Free Software Foundation" – это некоммерческая благотворительная организация, осуществляющая сбор средств на проект GNU. Целью проекта GNU является создание завершенной, бесплатной UNIX-подобной операционной системы. Многие из попутно разработанных инструментальных средств уже стали отраслевыми стандартами.
[URL 58] Web Server Information
www.netcraft.com/survey/servers.html
Ссылки на домашние страницы, находящиеся более чем на 50 web-серверах. Часть из них является коммерческими продуктами, другие же распространяются бесплатно.

Библиография

[Bak72] F.T. Baker. Chief programmer team management of production programming. IBM Systems Journal, 1 1(1):56—73, 1972.

[BBM96] V. Basili, L. Brand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(10):751-761, October 1996.

[Ber96] Albert J. Bernstein. Dinosaur Brains: Dealing with All Those Impossible People at Work. Ballantine Books, New York, NY, 1996.

[Bra95] Marshall Brain. Win32 System Services. Prentice Hall, Englewood Cliffs, NJ, 1995.

[Bro95] Frederick P. Brooks, Jr.. The Mythical Man Month: Essays on Software Engineering. Addison -Wesley, Reading, MA, anniversary edition, 1995.

[CG90] N. Carriero and D. Gelenter. How to Write Parallel Programs: A First Course. MIT Press, Cambridge, MA, 1990.

[CN91] Brad J. Cox and Andrex J. Novobilski. Object-Oriented Programming, An Evolutionary Approach. Addison-Wesley, Reading, MA, 1991.

[Coc97a] Alistair Cockburn. Goals and use cases. Journal of Object Oriented Programming, 9(7):35-40, September 1997.

[Coc97b] Alistair Cockburn. Surviving Object-Oriented Projects: A Manager's Guide. Addison Wesley Longman, Reading, MA, 1997.

[Cop92] James O. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading, MA, 1992.

[DL99] Tom Demarco and Timothy Lister. Peopleware: Productive Projects and Teams, Dorset House, New York, NY, second edition, 1999.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: Improving the Design of Existing Code. Addison Wesley Longman, Reading, MA, 1999

[Fow96] Martin Fowler. Analysis Patterns: Reusable Object Models. Addison Wesley Longman, Reading, MA, 1996.

[FS97] Martin Fowler and Kendall Scott. UML Distilled: Applying the Standard Object Modeling Language. Addison Wesley Longman, Reading, MA, 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[Gla99a] Robert L. Glass. Inspections – Some surprising findings. Communications of the ACM, 42(4): 17–19, April 1999.

[Gla99b] Robert L. Glass. The realities of software technology payoffs. Communications of the ACM, 42(2):74–79, February 1999.

[Hol78] Michael Holt. Math Puzzles and Games, Dorset Press, New York, NY, 1978.

[Jac94] Ivar Jacobson. Object-Oriented Software Engineering: A Use-Case Driven Approach. Addison-Wesley, Reading, MA, 1994.

[KLM+97] Gregor Kiczales, John Lamping, AnuragMendhekar, Chris Maeda, Cris-tina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In European Conference on Object-Oriented Programming (ECOOP), volume LNCS 1241. Springer-Verlag, June 1997.

[Knu97a] Donald Ervin Knuth. The Art of Computer Programming: Fundamental Algorithms, volume 1. Addison Wesley Longman, Reading, MA, third edition, 1997.

[Knu97b] Donald Ervin Knuth. The Art of Computer Programming: Seminumeri-cal Algorithms, volume 2. Addison Wesley Longman, Reading, MA, third edition, 1997.

[Knu98] Donald Ervin Knuth. The Art of Computer Programming: Sorting and Searching, volume 3.

Addison Wesley Longman, Reading, MA, second edition 1998.

[KP99] Brian W. Kernighan and Rob Pike. The Practice of Programming. Addison Wesley Longman, Reading, MA, 1999.

[Kru98] Philippe Kruchten. The Rational Unified Process: An Introduction. Addison Wesley Longman, Reading, MA, 1998.

[Lak96] John Lakos. Large-Scale C++Software Design. Addison Wesley Longman, Reading, MA, 1996.

[LH89] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. IEEE Software, pages 38–48, September 1989.

[Lis88] Barbara Liskov. Data abstraction and hierarchy. SIGPLAN Notices, 23(5), May 1988.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. Lex and Yacc. O'Reilly & Associates, Inc., Sebastopol, CA, second edition, 1992.

[McC95] Jim McCarthy. Dynamics of Software Development. Microsoft Press, Redmond, WA, 1995.

[Mey96] Scott Meyers. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Addison-Wesley, Reading, MA, 1996.

[Mey97a] Scott Meyers. Effective C++: 50 Specific Ways to Improve Your Programs and Designs. Addison Wesley Longman, Reading, MA, 1997.

[Mey97b] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.

[Pet 98] Charles Petzold. Programming Windows, The Definitive Guide to the Win32 API. Microsoft Press, Redmond, WA, fifth edition, 1998.

[Sch95] Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, New York, NY, 1995.

[Sed83] Robert Sedgewick. Algorithms. Addison-Wesley, Reading, MA, 1983.

[Sed 92] Robert Sedgewick. Algorithms in C++. Addison-Wesley, Reading, MA, 1992.

[SF96] Robert Sedgewick and Phillipc Flajolet. An Introduction to the Analysis of Algorithms. Addison-Wesley, Reading, MA, 1996.

[Ste92] W. Richard Stevens. Advanced Programming in the Unix Environment. Addison-Wesley, Reading, MA, 1992.

[Ste98] W. Richard Stevens. Unix Network Programming, Volume I: Networking APIs: Sockets andXti. Prentice Hall, Englewood Cliffs, NJ, 1998.

[Ste99] W. Richard Stevens. Unix Network Programming, Volume 2: Interprocess Communications. Prentice Hall, Englewood Cliffs, NJ, second edition, 1999.

[Str35] James Ridley Stroop. Studies of interference in serial verbal reactions. Journal of Experimental Psychology, 18:643–662, 1935.

[WK82] James Q. Wilson and George Kelling. The police and neighborhood safety. The Atlantic Monthly, 249(3):29–38, March 1982.

[YC86] Edward Yourdon and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, Englewood Cliffs, NJ, second edition, 1986.

[You95] Edward Yourdon. Managing projects to produce good-enough software. IEEE Software, March 1995.

Приложение B
Ответы к упражнениям

Упражнение 1 из раздела "Ортогональность"

Ответ: По нашему разумению, более ортогональным является класс Split2. Он сосредоточен на собственной задаче – расщеплении строк и игнорирует подробности, связанные с источником обрабатываемых им строк. Это не только упрощает разработку программы, но и придает ей большую гибкость. Класс Split2 может расщеплять строки, считываемые из файла, сгенерированные другой программой или передаваемые через операционную среду.

Упражнение 2 из раздела "Ортогональность"

Ответ: Если все сделано корректно, то, по всей вероятности, немодальное. Система, которая использует немодальные диалоговые окна, испытывает меньшее беспокойство о том, что происходит в любой конкретный момент времени. Скорее всего, она будет обладать лучшей инфрастуктурой взаимодействия между модулями по сравнению с модальной системой, которая может содержать встроенные предположения о состоянии системы – предположения, которые могут привести к большему связыванию и уменьшению ортогональности.

Упражнение 3 из раздела "Ортогональность"

Ответ: Здесь есть элемент лукавства. Объектная технология может обеспечить наличие более ортогональной системы, но поскольку она имеет больше средств, которые могут эксплуатироваться с нарушением режима, в реальности легче создать неортогональную систему, используя объекты, чем создавать ее при помощи процедурного языка. Ее особенности – множественное наследование, исключительные ситуации, перегрузка операторов и переопределение родительского метода (через механизм подклассов) – предоставляют достаточные возможности для увеличения связанности не столь очевидными способами.

Применяя объектную технологию и приложив небольшое дополнительное усилие, вы можете добиться наличия более ортогональной системы. И хотя вы всегда можете написать неструктурированную программу на процедурном языке, объектно-ориентированные языки, используемые в малых дозах, могут сделать ее более насыщенной.

Упражнение 4 из раздела "Прототипы и памятные записки"

Ответ: Для спасения ситуации прибегнем к устаревшим технологиям! Нарисуйте на лекционной доске несколько картинок – автомобиль, телефон и дом – с помощью фломастера. Для этого не нужно быть великим художником, вполне достаточно условных изображений. Поместите на доску памятные записки, описывающие содержимое целевых страниц в активных областях экрана. В ходе встречи вы можете совершенствовать рисунки и менять расположение

памятных записок.

Упражнение 5 из раздела "Языки, отражающие специфику предметной области"

Ответ: Поскольку мы хотим, чтобы язык был расширяемым, сделаем таблицу синтаксического анализатора управляемой. Каждый элемент таблицы содержит символ команды, флаг, говорящий о необходимости аргумента, и имя подпрограммы, вызываемой для обработки этой конкретной команды.

```
typedef struct {
char cmd; /* the command letter */
int hasArg; /* does it take an argument */
void (*func)(int, int); /* routine to call */
} Command;
static Command cmds[] = {
{'P', ARG, doSelectPen},
('V', NO_ARG, doPenUp},
{'D', NO_ARG, doPenDown},
{'N,' ARG, doPenDir},
{'E', ARG, doPenDir},
{'S', ARG, doPenDir},
{'W', ARG, doPenDir}
};
```

Основная программа довольно проста: считать строку, отыскать команду, при необходимости принять аргумент, затем вызвать функцию обработчика.

```
while (fgetsfbuff, sizeof(buff), stdin)) {
Command *cmd = findCommand(*buff);
if (cmd) {
  int arg = 0;
  if (cmd->hasAr&& !getArg(buff+1, &arg)) {
     fprintf(stderr,"'%c' needs an argument\n", *buff);
     continue;
  }
cmd->func(*buff, arg);
}
}
```

Функция, которая ищет команду, исполняет последовательный перебор таблицы, возвращая либо совпадающий элемент, либо NULL.

```
Command *findCommand(int cmd) {
int i;
for (i = 0; i<ARRAY.SIZE(cmds); i++) {
  if (cmds[i].cmd==cmd)
    return cmds + i;
  }
fprintf(stderr, "Unknown command %c'\n", cmd);
return 0;
}
```

И наконец, считывание числового аргумента довольно просто, если использовать подпрограмму sscanf.

```
int getArg(const char *buff, int 'result) {
  return sscanf(buff, "%d", result) == 1;
}
```

Упражнение 6 из раздела "Языки, отражающие специфику предметной области"

Ответ 6: При использовании BNF спецификация времени могла бы выглядеть следующим образом:

```
<tlme>::= <hour> <ampm> |
        <hour>: <minute> <ampm> |
        <hour>: <minute>
<ampm>::= am|pm
<hour>::=<digit> |
        <digit>::=<digit>
<mlnute>::=<digit><digit>
<digit>::= 0|1|2|3|4|5|6|7|8|9
```

Упражнение 7 из раздела "Языки, отражающие специфику предметной области"

Ответ: В нашем примере мы составили программу, используя генератор bison, который представляет собой GNU-версию генератора уасс. Для ясности здесь показано только тело программы синтаксического анализатора. Полная версия есть на сайте www.pragmaticprogrammmer.com.

```
time: spec EOF
{ if ($1>= 24*60) yyerror("Time is too large");
  printf("%d minutes past midnight\n", $1);
  exit(0);
}
;
spec:  hour ':' minute
    { $$ = $1 + $3;
    }
 | hour ':' minute ampm
{ if ($1>11*60) yyerrorf "Hour out of range");
  $$ = $1 + $3 + $4;
}
| hour ampm
  {if ($1>11*60) yyerror("Hour out of range");
   $$ = $1 + $2;
   }
;
hour: hour_num
    {if ($1>23) yyerror("Hour out of range");
```

```
    $$ = $1 * 60;
   };
minute: DIGIT DIGIT
    {$$ = $1*10 + $2;
    if ($$> 59) yyerrorf "minute out of range") ,
    };
ampm: AM {$$ = AM_MINS;}
   |     PM {$$ = PM_MINS;)
   ;
hour num:  DIGIT {$$ = $1;)
   | DIGIT DIGIT {$$ = $1*10 + $2;}
   ;
```

Упражнение 8 из раздела "Языки, отражающие специфику предметной области"

Ответ:

```
$_ = shift;
/"(\d\d?)(am|pm)$/ && doTime($1, 0, $2, 12);
/"(\d\d?):(\d\d)(am|pm)$/ && doTime($1, $2, $3, 12);
/"(\d\d?):(\d\d)$/ && doTime($1, $2, 0, 24);
die "Invalid time $_\n";
#
# doTime(hour, min, ampm, maxHour)
#
sub doTime($$$$) {
my ($hour, $min, $offset, $maxHour) = @_;
die "Invalid hour: $hour" if ($hour>= $maxHour);
$hour += 12 if ($offset eq "pm")
print $hour*60 + $min, " minutes past midnight\n";
exit(0);
}
```

Упражнение 9: из раздела "Оценка"

Ответ: Ответ должен быть изложен, исходя из нескольких допущений:

• Лента содержит информацию, которую необходимо передать.

• Известна скорость ходьбы человека.

• Известно расстояние между компьютерами.

• Временем, необходимым для переноса информации на ленту и с ленты, можно пренебречь.

• Потери данных при хранении на ленте примерно равны их потерям при передаче по каналу связи.

Упражнение 10 из раздела "Оценка"

Ответ: Учитывая допущения ответа 9: Объем информации, содержащейся на стриммерной кассете (4 Гбайт), составляет 32 x 10^9 бит, так что передача эквивалентного объема по каналу со скоростью 1 Мбайт/с заняла бы около 32000 сек. (примерно 9 ч). Если человек движется с постоянной скоростью 3,5 мили в час, то, для того чтобы канал связи превзошел курьера, два компьютера должны располагаться друг от друга на расстоянии не менее 31 мили. Если это расстояние меньше, то победа остается за человеком.

Упражнение 11 из раздела "Обработка текста"

Ответ: Ответ к данному упражнению составлен на языке Perl.

```
my @consts;
my $name = <>;
die "Invalid format – missing name" unless defined($name);
chomp $name;
# Read in the rest of the file
while (<>) {
  chomp;
  s/"\s*//; s/\s*$//;
  die "Invalid line: $_" unless /"(\w+)$/;
  push @consts, $_;
}
# Now generate the file
open(HDR, ">$name.h") or die "Can't open $name.h: $!";
open(SRC, ">$name.c") or die "Can't open $name.c: $!";
my $uc_name = uc($name);
print HDR "/* File generated automatically – do not edit */\n";
print HDR "extern const char *$ {ucjiame)_name[];";
print HDR "typedef enum {\n"; print HDR join",\n", @consts;
print HDR "\n) $uc_name;\n\n";
print SRC "* File generated automatically – do not edit */\n";
print SRC "const char *$ {uc name}_name[] = {\n \"";
print SRC join "\",\n \"", @consts;
print SRC "\"\n};\n";
close(SRC);
close(HDR);
```

Используя принцип DRY, мы не будет вырезать и вклеивать этот вновь написанный файл в нашу программу. Вместо этого мы «включим» его – данный плоский файл является главным источником этих констант. Поэтому нам понадобится файл сборки для восстановления заголовка при изменении файла. Следующий фрагмент содержится в системе отладки в исходном дереве (имеется на web-сайте).

```
etest.c etest.h: etest.inc enumerated.pl
        perl enumerated.pl etest.inc
```

Упражнение 12 из раздела "Обработка текста"

Ответ: Вот ответ, написанный на языке Perl.

```
my $dir = shift or die "Missing directory" ,
for my Sfile (glob(u$dir/*.pr)) {
open(IP, "$file") or die "Opening $file: $!";
undef $/; # Turn off input record separator -
my Scontent = <IP>; # read whole file as one string.
close(IP);
if (Scontent Γrusestrict/m) {
rename Sfile, "$file.bak" or die "Renaming $file: $!"; open(OP, ">$file") or die "Creating $file: $!";
# Put 'use strict' on first line that
# doesn't start #
Scontent =" sr(V.#)/\nuse strict;\n\n/m',
print OP Scontent; close(OP);
print "Updated $file\n";
)
else {
print "Sfile already strict\n":
)
}
```

Упражнение 13 из раздела "Генераторы исходных текстов"

Ответ: Решение реализовано на языке Perl. В программе происходит динамическая загрузка модуля для генерации требуемого языка, так что добавление новых языков не представляет труда. Главная программа загружает внутреннюю часть (основанную на параметре командной строки), затем считывает ее входные данные и вызывает подпрограммы генерации текста, основанные на содержимом каждой из строк. Мы особенно не суетимся, если речь идет об обработке ошибок: если что-то не так, узнаем об этом довольно быстро.

```
my lang = shift or die "Missing language";
$lang .= "_cg.pm";
require <$lang> or die "Couldn't load $lang";
# Read and parse the file
my $name;
while (<>) {
  chomp;
if (/^ \s*$/)   {CG::blankLine();)
elsif ((/^ \#(.*)/)   {CG::comment($1);}
elsif ((/^M\s*(.+)/)   {CG::startMsg($1); $name = $1;}
elsif ((/^E/)    {CG::endMsg($name);}
elsif (/^F\s*(\w+)\s+(\w+)$/) {CG::simpleType($1,$2);}
elsif (/^F\s*(\w+)\s+(\w+)\[(\d+)\]$/) {CG::arrayType $1,$2,$3);}
else {
die "Invalid line: $ ";
}
}
```

Написание языковой серверной части не составит труда: создайте модуль, который

реализует шесть точек входа. Вот генератор текста на языке C:

```
#!/usr/bin/perl – w
package CG;
use strict;
# Code generator for 'C' (see cg_base.pl)
sub blankLine()  {print "\n"; }
sub comment()  {print "/*$_[0] */\n"; }
sub startMsg()   {print "typedef struct {\n"; }
sub endMsg()    {print "} $_[0];\n\n"; }
sub arrayType() {
  my ($name, $type, $size) = @_;
  print " $type $name\[$size];\n":
}
sub simpleType{) {
  my ($name, $type) = @_;
  print " $type $пате;\п";
}
1;
```

А вот генератор текста на языке Pascal:

```
#!/usr/bin/perl – w
package CG;
use strict;
# Code generator for 'Pascal' (see cg_base.pl)
sub blankLine() {print "V;)
sub comment() {print "{$_[0] }\n";)
sub startMsg()  {print "$_[0] = packed record\n"; }
sub endMsg()   {print "end;\n\n"; }
sub arrayType() {
  my ($name, $type, $size) = @_;
  $size--;
  print" $name: array[0…$size] of $type;\n";
)
sub simpleType() {
  my ($name, $type) = @_;
  print" $name: $type;\n7 ',
}
1;
```

Упражнение 14 из раздела "Проектирование по контракту"

Ответ: Этот пример на языке Eiffel удачен. Мы требуем передачи непустых данных и гарантируем, что семантика циклического двунаправленного списка будет соблюдена. Это также способствует нахождению сохраненной строки. Поскольку это некий отложенный класс, то действительный класс, его реализующий, допускает использование любого основного

механизма по своему усмотрению. Он может использовать указатели или массив, или что-то еще; пока он соблюдает контракт, мы не беспокоимся.

Упражнение 15 из раздела "Проектирование по контракту"

Ответ: Это неудачно. Математическое действие в индексном выражении (index-1) не будет работать с граничными условиями, подобными первой точке входа. Постусловие предполагает определенную реализацию; контракты должны быть более абстрактными по сравнению с указанным выше.

Упражнение 16 из раздела "Проектирование по контракту"

Ответ 16: Это удачный контракт, но неудачная реализация. Здесь высовывает свою уродливую голову ошибка типа "Heisenbug" [URL52]. Вероятно, программист допустил опечатку – набрал pop вместо top. Хотя это простой и надуманный пример, весьма трудно диагностировать побочные эффекты в утверждениях (или в любом, самом неожиданном месте в программе).

Упражнение 17 из раздела "Проектирование по контракту"

Ответ: Мы продемонстрируем функциональные сигнатуры на языке Java, обозначая предусловия и постусловия в соответствии с iContract. Сначала инвариант для класса:

```
/**
* @invariant getSpeed() >0
* implies isFull() // Не запускать пустое
* @invariant getSpeed()>=0 &&
* getSpeed() <10 // Проверка границ
*/
```

Затем предусловия и постусловия:

```
/**
* @pre Math.abs(getSpeed() – x) <= 1 // Единственный выбор
* @pre x>= 0 &&  x > 10  // Проверка границ
* @post getSpeed() == x    // Проверка скорости
*/
public void setSpeed (final int x)
/**
* @pre !isFull()  // Незаполнять дважды
* @post isFull() // Убедитесь, что было выполнено
*/
void fill()
/**
* @pre isFull()    // Не очищатьдважды
* @post !isFull() // Убедиться, что выполнено
*/
```

void empty()

Упражнение 18 из раздела "Проектирование по контракту"

Ответ: В этом ряду содержится 21 число. Если вы ответили «20», то допустили так называемую ошибку "поста охраны".

Упражнение 19 из раздела "Программирование утверждений"

Ответ:

1. В сентябре 1752 г. было всего лишь 19 дней. Это было сделано с целью синхронизации при переходе с юлианского на григорианский календарь.
2. Каталог мог быть удален другим процессом, у вас нет прав доступа на его чтение, выражение &sb может быть недопустимым – вы все уловили.
3. Мы проявили малодушие, не указав типов a и b. Во время перегрузки операторов могло случиться так, что поведение знаков +, =, или != стало непредсказуемым. Кроме того, a и b могут быть псевдонимами одной и той же переменной, так что при втором присвоении произойдет перезапись значения, сохраненного во время первого.
4. В неевклидовой геометрии сумма углов треугольника не будет составлять 180°. Подумайте о треугольнике, отображаемом на поверхности сферы.
5. Минуты, приходящиеся на високосный год, могут состоять из 61 или 62 секунд.
6. Переполнение может оставить результат операции a+1 отрицательным (это также может произойти в языках C и C++).

Упражнение 20 из раздела "Программирование утверждений"

Ответ: Мы решили реализовать очень простой класс с единственным статическим методом TEST, который выводит на печать сообщение и след стека, если переданный параметр condition является ложным.

```
package com.pragprog.util;
import java.lang.System; //для exit()
import java.lang.Thread; //для dumpStack()
public class Assert {
/** Write a message, print a stack trace and exit if
   * our parameter is false.
   */
public static void TEST(boolean condition) {
  if (Icondition) {
    System.out.println("==Assertion Failed==");
    Thread.dumpStack();
     System.exit(1);
   }
}
```

```
// Testbed. If our argument is 'okay', try an assertion that
// succeeds, if 'fail' try one that fails
public static final void main(String args[]) {
if (args[0].compareTo("okay") == 0) {
   TEST(1 == 1);
}
else if (args[0].compareTo("fail") == 0) {
   TEST(1 == 2);
}
else {
  throw new RuntimeException("Bad argument") ,
  }
 }
}
```

Упражнение 21 из раздела "Случаи, когда используются исключения"

Ответ: Нехватка памяти является исключительным состоянием, поэтому мы полагаем, что в случае (1) должно возбуждаться исключение.

Невозможность отыскания точки входа – вполне нормальная ситуация. Приложение, которое вызывает наш класс-набор, может написать программу, которая проверяет наличие точки входа, перед тем как добавить потенциальный дубликат. Мы полагаем, что в случае (2) нужно просто осуществить возврат ошибки.

Случай (3) более проблематичен – если указатель null играет существенную роль в приложении, его добавление к контейнеру может быть оправдано. Но если для хранения пустых значений нет веских оснований, то, по всей вероятности, необходимо возбудить исключительную ситуацию.

Упражнение 22 из раздела "Балансировка ресурсов"

Ответ: В большинстве реализаций языков C и C++ отсутствуют способы проверки того, что указатель действительно указывает на допустимый блок памяти. Обычная ошибка состоит в освобождении блока памяти и организации ссылки на этот блок далее в тексте программы. К тому времени этот блок памяти уже может быть перераспределен для других целей. Обнуляя указатель, программисты надеются предотвращать эти инородные ссылки – в большинстве случаев разыменование указателя null генерирует ошибку в ходе выполнения программы.

Упражнение 23 из раздела "Балансировка ресурсов"

Ответ: Обнуляя ссылку, вы уменьшаете число указателей на упомянутый объект на единицу. Как только этот счетчик становится равным нулю, объект получает право на сбор «мусора». Обнуление ссылок может играть существенную роль в продолжительных по времени программах, где программистам приходиться удостоверяться, что использование памяти со временем не возрастает.

Упражнение 24 из раздела "Несвязанность и закон Деметера"

Ответ: Файл заголовка предназначен для определения интерфейса между соответствующей реализацией и внешним миром. Сам по себе файл заголовка не обязан обладать информацией о внутренней организации класса Date – от него лишь требуется сообщить компилятору о том, что конструктор принимает класс Date в качестве параметра. Поэтому, если файл заголовка не использует Dates в подставляемых функциях, второй фрагмент будет работать просто замечательно.

А что же с первым фрагментом? Если он используется в небольшом проекте, то все нормально, за исключением того, что вы без особой надобности заставляете все элементы программы, которые используют класс Person1, также включать файл заголовка для класса Date. Как только подобное употребление становится обычной практикой в неком проекте, вскоре обнаружите, что включение одного файла заголовка заканчивается включением большей части системы, что существенно увеличивает время компиляции.

Упражнение 25 из раздела "Несвязанность и закон Деметера"

Ответ: Переменная acct передается в виде параметра, так что вызов getBalance является допустимым. Вызов amt.printFormat() таковым не является. Мы не «владеем» amt, и он не был передан нам. Мы могли устранить связывание showBalance с Money при помощи вставки, подобной представленной ниже:

```
void showBalance(BankAccount b) {
   b.printBalance();
}
```

Упражнение 26 из раздела "Несвязанность и закон Деметера"

Ответ: Поскольку класс Colada создает и владеет myBlender и myStuff, то обращения к addIngredients и elements являются допустимыми.

Упражнение 27 из раздела "Несвязанность и закон Деметера"

Ответ: В этом случае processTransaction владеет amt – он создается на стеке. Происходит передача acct, поэтому допустимыми являются как setValue, так и setBalance. Но processTransaction не владеет who, поэтому вызов who->name() является нарушением. Закон Деметера предлагает заменить эту строку на следующую:

```
markWorkflow(acct.name (), SET_BALANCE);
```

Программе в processTransaction не придется узнавать, какой дочерний объект в пределах BankAccount носит это имя – эта информация о структуре не должна разглашаться через контракт BankAccount. Вместо этого мы запрашиваем у BankAccount имя на счете. Он знает, где хранится имя (может быть, в объекте Person, в объекте Business, или в полиморфном объекте Customer).

Упражнение 28 из раздела "Метапрограммирование"

Ответ: Здесь не приводятся категорические ответы – вопросы предназначались в основном для того, чтобы дать вам пищу для размышлений. И вот что мы думаем:

1. Назначения коммуникационного порта. Ясно, что эта информация должна сохраняться в виде метаданных. Но на каком уровне детализации? Некоторые коммуникационные программы системы Windows позволяют выбирать только скорость в бодах и порт (скажем, COM1 – COM4). Но вероятно вам придется указать размер слова, четность, стоповые биты, и настройку дуплексной связи. Старайтесь допускать самый мелкий уровень детализации, там где это разумно с практической точки зрения.

2. Поддержка выделения синтаксических конструкций различных языков в программе редактирования. Она должна быть реализована в виде метаданных. Вы же не хотите, чтобы вам пришлось переделывать программу только потому, что в последней версии языка Java было введено новое ключевое слово.

3. Поддержка редактора для различных графических устройств. Эту поддержку было бы трудно реализовать исключительно в виде метаданных. Вам не хотелось бы нагружать приложение многими драйверами устройств только для того, чтобы выбрать один из них во время выполнения программы. Однако вы могли воспользоваться метаданными для указания имени драйвера и динамической загрузки программы. Это еще один аргумент для сохранении метаданных в удобочитаемом формате; если вы используете программу для установки дисфункционального видеодрайвера, то вы не сможете переустановить его, пользуясь этой программой.

4. Конечный автомат для программы синтаксического анализа или сканера.

Это зависит от того, анализируете вы или просматриваете. Если анализируете некоторые данные, которые жестко определены в стандартах и скорее всего не будут изменены без одобрения конгресса США, то жесткое кодирование вполне годится. Но если вы сталкиваетесь с более изменчивой ситуацией, то может быть, более выгодным является внешнее определение таблиц состояний.

5. Типовые значения и результаты, используемые в модульном тестировании.

Большинство приложений определяет эти значения как встроенные в тестовый стенд, но вы можете добиться большей гибкости, перемещая тестовые данные и определение приемлемых результатов за пределы самой программы.

Упражнение 29 из раздела "Всего лишь представление"

Ответ: В программу Flight мы добавим ряд дополнительных методов душ поддержания двух списков «слушателей»: для уведомления о листе ожидания и о полной загрузке рейса.

```
public interface Passenger {
  public void waitListAvailable();
}
public interface Flight {
...
  public void addWaitListListener(Passenger p);
  public void removeWaitUstUstener(Passenger p);
  public void addFullListener(FullListener b);
```

```
  public void removeFullListener(FullListener b);
...
}
public interface BigReport extends FullListener {
  public void FlightFullAlert(Flight f);
}
```

При неудачной попытке добавить Passenger, поскольку рейс полностью забронирован, мы можем (как вариант) поместить Passenger в лист ожидания. При открытии вакансии производится вызов метода waitListAvailable. Затем этот метод может осуществить выбор: либо добавить Passenger автоматически, либо дать указание сотруднику авиакомпании позвонить заказчику и выяснить, заинтересован ли он еще в рейсе, и т. п. Теперь мы обладаем достаточной гибкостью, чтобы избрать линию поведения, исходя из пожеланий клиента.

Кроме того, мы хотим избежать ситуаций, при которых BigReport разбирает тонны записей, отыскивая полностью забронированные рейсы. Зарегистрировав BigReport в качестве «слушателя» Flights, каждый индивидуальный Flight может сообщать, когда он полностью (или почти полностью) забронирован. Теперь пользователи могут мгновенно получить оперативные, с точностью до минуты, сообщения из BigReport, а не ожидать часами окончания его работы, как это было раньше.

Упражнение 30 из раздела "Доски объявлений"

Ответ:

1 . **Обработка изображения.** Для простого распределения рабочей нагрузки между параллельными процессами более чем адекватной может оказаться общедоступная очередь работ. Вы можете рассмотреть систему "доска объявлений" при наличии обратной связи, т. е. если результаты обработки одного фрагмента изображения влияют на другие фрагменты так, как это происходит в системах искусственного зрения или сложных трехмерных преобразованиях изображений.

2 . **Календарное планирование для групп.** Для этого "доска объявлений" очень даже пригодится. Вы можете поместить назначенные собрания и готовность на "доску объявлений". Есть объекты, функционирующие автономно; в данном случае очень важна обратная связь, а участники могут приходить и уходить.

Можно рассмотреть возможность разделения "доски объявлений" в зависимости от того, кто осуществляет поиск: младший персонал может заботиться только о локальном офисе, отдел кадров интересоваться только англо-говорящими офисами во всем мире, а исполнительный директор – всем сразу.

В форматах данных имеется некий элемент гибкости: мы можем проигнорировать форматы или языки, которых не понимаем. Нам придется понимать различные форматы только для тех офисов, которые встречаются друг с другом, и не придется подвергать всех участников полному транзитивному замыканию всевозможных форматов. При этом связанность уменьшается там, где нужно, и мы не имеем искусственных ограничений.

3 . **Средство мониторинга компьютерной сети.** Это весьма сходно с программой обработки заявлений на ипотечный кредит/ссуду, описанной в примере приложения (с. 153). На доску помещаются сообщения о неисправностях, присылаемые пользователями, и автоматические генерируемые статистические данные. Сотрудник (или программный агент) может анализировать "доску объявлений", чтобы осуществлять диагностику неисправностей в

сети: две ошибки в линии могут быть отнесены на счет космических лучей, но 20000 ошибок говорят о проблеме в аппаратном обеспечении. Подобно детективам, разгадывающим тайну убийства, вы можете использовать множественные объекты, анализируя и внося свою лепту в решение проблем, связанных с компьютерной сетью.

Упражнение 31 из раздела "Программирование в расчете на стечение обстоятельств"

Ответ: Эта программа представляет ряд потенциальных проблем. Во-первых, она предлагает наличие текстовой среды. Это, может быть, и прекрасно, если предположение истинно, но что, если эта программа вызывается из графической среды, где не открыты ни stderr, ни stdin?

Во-вторых, есть проблематичный оператор gets, который будет записывать столько символов, сколько он получит в переданный буфер. Злонамеренные пользователи использовали это, когда им не удавалось проделать бреши типа buffer overrun в защите многих различных систем. Никогда не пользуйтесь gets().

В-третьих, программа предполагает, что пользователь понимает английский язык.

И наконец, никто, находясь в здравом уме, не станет прятать средство взаимодействия с пользователем в недра библиотечной подпрограммы.

Упражнение 32 из раздела "Программирование в расчете на стечение обстоятельств"

Ответ: Работа программы strcpy в системе POSIX не гарантируется при наличии перекрывающихся строк. С некоторыми архитектурами она, случается, и работает, но лишь при стечении определенных обстоятельств.

Упражнение 33 из раздела "Программирование в расчете на стечение обстоятельств"

Ответ: Она не будет работать в контексте апплета при наличии ограничений доступа по записи на локальный диск. Если вы можете выбирать, работать ли через графический интерфейс или нет, то, вероятно, захотите осуществить динамический анализ текущей среды. В этом случае вы наверняка захотите поместить файл журнала вне локального диска, если к нему нет доступа.

Упражнение 34 из раздела "Скорость алгоритма"

Ответ: Ясно, что мы не можем давать никаких абсолютных ответов к этому упражнению. Однако можем дать вам пару намеков.

Если ваши результаты не ложатся на гладкую кривую, то вы захотите проверить, не используется ли мощность вашего процессора каким-либо другим процессом. По всей вероятности, вы не получите хороших показателей в многопользовательской системе, и даже если окажетесь единственным пользователем, можете заметить, что фоновые процессы периодически отбирают циклы у ваших программ. Вы также можете захотеть проверить использование памяти: если приложение начинает использовать область свопинга, то производительность резко снижается.

Интересно поэкспериментировать с различными компиляторами и установочными параметрами оптимизации. Мы обнаружили, что весьма впечатляющее ускорение стало возможно, благодаря использованию агрессивной оптимизации. Мы также обнаружили, что на более распространенных архитектурах типа RISC компиляторы фирмы изготовителя часто превосходили по быстродействию более переносимую GCC. Возможно, изготовитель посвящен в тайны эффективной генерации программ на этих машинах.

Упражнение 35 из раздела "Скорость алгоритма"

Ответ: Программа printTree использует приблизительно 1000 байт стекового пространства для буферной переменной. Для движения вниз по древовидной схеме она рекурсивно вызывает саму себя, и каждый вложенный вызов добавляет еще 1000 байт к стеку. Она также вызывает саму себя, когда добирается до вершин, но заканчивает работу сразу, как только обнаружит, что переданный указатель обнулен. Если глубина дерева равна D, то максимальный объем, необходимый стеку, составляет (грубо) 1000 x(D+ 1).

Сбалансированное двоичное дерево содержит вдвое больше элементов на каждом уровне. Дерево глубиной D содержит 1+2+4+8 +… +2^(D-1), или 2^D – 1 элементов. Следовательно, дереву, состоящему из миллиона элементов, будет необходимо [lg(1000001], или 20 уровней.

Поэтому мы рассчитываем, что наша подпрограмма будет использовать примерно 21000 байт стекового пространства.

Упражнение 36 из раздела "Скорость алгоритма"

Ответ: На ум приходит несколько процедур оптимизации. Программа printTree вызывает саму себя на вершинах дерева лишь для того, чтобы закончить работу при отсутствии потомков. Этот вызов увеличивает максимальную глубину стека примерно на 1000 байт. Можно также исключить рекурсию хвоста (второй рекурсивный вызов), хотя это и не будет затрагивать использование стека в наихудшем случае.

```
while (node) {
  if (node->left) printTree(node->left);
  getNodeAsString(node, buffer);
  puts(buffer);
  node = node->right;
}
```

Но самая большая выгода возникает при назначении одного-единственного буфера, доступ к которому осуществляется при всех обращениях к printTree. Если передать этот буфер в виде параметра для рекурсивных обращений, то будет назначено всего 1000 байт вне зависимости от глубины рекурсии.

```
void printTreePrivate(const Node *node, char *buffer) {
  if (node) {
    printTreePrivate(node->!eft, buffer);
    getNodeAsStringfnode, buffer);
    puts(buffer);
    printTreePrivate(node->right, buffer);
  }
```

```
}
void newPrintTree(const Node *node) {
  char buffer[1000];
  printTreePrivate(node, buffer);
)
```

Упражнение 37 из раздела "Скорость алгоритма"

Ответ: Это можно сделать двумя путями. Один из них заключается в том, чтобы перевернуть проблему с ног на голову. Если в массиве есть лишь один элемент, мы не осуществляем итерации в цикле. Каждая дополнительная итерация удваивает размер массива, в котором можно осуществлять поиск. Отсюда общая формула размера массива: n = 2^m, где m – число итераций. Если прологарифмировать обе части с основанием 2, получим выражение lg(n) = lg(2^m), которое из определения логарифма превращается в lg(n) =m.

Упражнение 38 из раздела "Реорганизация"

Ответ: Здесь мы могли бы предложить весьма умеренную реструктуризацию: убедитесь, что каждый тест выполняется лишь один раз, и сделайте все вычисления стандартными. Если выражение 2*basis (…)* 1.05 появляется в других местах программы, то, вероятно, его стоит сделать функцией. В данном случае мы этого делать не стали.

Мы добавили массив rate_lookup, инициализированный таким образом, что элементам, отличным от Texas, Ohio и Maine, присвоено значение 1. Этот подход облегчает добавление значений для других штатов в будущем. В зависимости от ожидаемой схемы использования мы могли бы сделать поле points также средством поиска в массиве.

```
rate = rate_lookup[state];
amt = base * rate;
calc = 2*basis(amt) + extra(amt)*1.05;
if (state == OHIO)
   points = 2;
```

Упражнение 39 из раздела "Реорганизация"

Ответ: Когда вы видите, что кто-либо использует перечислимые типы (или эквивалентные им в языке Java), для того чтобы провести различие между вариантами некоего типа, во многих случаях вы можете улучшить программу за счет создания подклассов:

```
public class Shape {
  private double size;
  public Shape(double size) {
     this.size = size;
  }
  public double getSize() {return size;)
}
public class Square extends Shape {
```

```
  public Square(double size) {
    super(size);
  }
  public double area() {
    double size = getSize();
    return size*size;
  }
)
public class Circle extends Shape {
  public Circle(double size) {
    super(size);
  }
  public double area() {
    double size = getSize();
    return Math.PI*size*size/4.0;
  }
}
// efc…
```

Упражнение 40 из раздела "Реорганизация"

Ответ: Этот случай интересен. На первый взгляд, кажется разумным, что у окна должна быть ширина и высота. Однако стоит подумать о будущем. Представим, что мы хотим обеспечить поддержку окон произвольной формы (что будет весьма трудно, если класс Window обладает всей информацией о прямоугольниках и их свойствах).

Мы бы предложили абстрагировать форму окна из самого класса Window.

```
public abstract class Shape {
//...
  public abstract boolean overlaps (Shape s);
  public abstract int getArea();
}
public class Window {
  private Shape shape;
  public Window(Shape shape)  {
     this.shape = shape;
     ...
}
public void setShape(Shape shape) {
   this.shape = shape;
   ...
}
public boolean overlaps(Window w) {
   return shape.overlaps(w.shape);
}
public int getArea() {
   return shape.getArea();
```

```
}
}
```

Заметим, что в этом подходе мы предпочли делегирование созданию подклассов: окно не является «разновидностью» формы – окно «имеет» форму. Оно использует форму для выполнения своей работы. Вы убедитесь, что во многих случаях делегирование оказывается полезным для реорганизации.

Мы могли бы расширить этот пример, внедрив интерфейс Java, указывающий на методы, которые должны поддерживаться неким классом для поддержания функций формы. Эта удачная идея означает, что, когда вы расширяете принцип формы, компилятор предупредит вас о классах, которые вы затронули. Мы рекомендуем использовать интерфейсы подобным способом при делегировании всех функций какого-либо другого класса.

Упражнение 41 из раздела "Программа, которую легко тестировать"

Ответ: Вначале добавим подпрограмму main, которая будет действовать как ведущий элемент модульного тестирования. В качестве аргумента она примет простой мини-язык: <E> будет означать опорожнение блендера, <F> – его наполнение, цифры 0–9 будут задавать скорость вращения ротора, и т. д.

```
public static void main(String args[]) {
// Create the blender to test
  dbc_ex blender = new dbc_ex();
// And test it according to the string on standard input
  try {
    int a;
    char c;
    while ((a = System.in.read())!= -1) {
      c = (char)a;
      if (Character.isWhitespace(c)) {
        continue;
      }
      if (Character.is Digit(c)) {
        blender.setSpeed(Character.digit(c, 10));
      }
     else {
       switch (c) {
        case 'F': blender.fi!l();
                  break;
        case 'E': blender.empty();
                  break;
        case 's': System.out.printlnfSPEED: " + blender.getSpeed());
                  break;
        case 'f': System.out.println(<FULL> + blender.isFull());
                  break;
        default: throw new RuntimeException(
                  "Unknown Test directive");
  }
```

```
 }
}
catch (java.io.lOException e) {
  System.err.println("Tesf jig failed: " + e.getMessage());
}
System.err.println("Completed blending\n");
System.exit(0);
}
```

Затем появится сценарий оболочки для управления тестированием.

```
#!/bin/sh
CMD="java dbc.dbc_sx"
failcount=0
expect okay() {
  if echo "$*" | $CMD #>/dev/null 2>&1
  then
  ...
  else
     echo "FAILED! $*"
     failcount='expr $failcount + 1'
  fi
}
expect_fail() {
if echo "$*" | SCMD>/dev/null 2> &1
then
  echo "FAILED! (Should have failed): $*"
   failcount='expr $failcount + 1'
fi
}
report() {
  if [$failcount -gt 0]
  then
     echo – e "\n\n*** FAILED $failcount TESTS\n"
     exit 1 # In case we are part of something larger
  else
    exit 0 # In case we are part of something larger
  fi
}
#
# Start the tests
#
expect_okay F12345678987654321OE # Should run thru
expect_fail F5 # Fails, speed too high
expect_fail 1 # Fails, empty
expect_fail F10E1 # Fails, empty
expect_fail F1238 # Fails, skips
expect_okay FE # Never turn on
```

expect_fail F1E # Emptying while running
expect_okay F10E # Should be ok
report # Report results

При тестировании проверяется, не имеют ли место недопустимые переходы в скорости вращения ротора, если вы пытаетесь опорожнить работающий блендер, и т. д. Мы помещаем эту процедуру в сборочный файл, так что можно провести компиляцию и запустить регрессионный тест, просто введя команду:

% make % make test

Обратите внимание на то, что мы выходим из процедуры тестирования с 0 иди 1, так что можем использовать это и как часть более обширного теста.

В требованиях ничего не говорилось о запуске данного компонента через сценарий или даже с использованием языка. Конечные пользователи этого не увидят. Но у нас есть мощный инструмент, который можно использовать для быстрого и исчерпывающего тестирования нашей программы.

Упражнение 42 из раздела "Ошибка в определении требований" Ответ:

1. Эта инструкция похожа на реальное требование: имеются ограничения, налагаемые на приложение со стороны операционной среды.

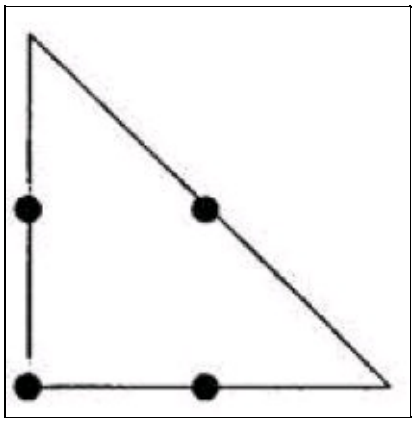
2. Это может быть корпоративным стандартом, но не требованием. Его лучше сформулировать так: "Фон диалогового окна должен настраиваться конечным пользователем. При поставке заказчику цвет будет серым". Еще лучше было бы сформулировать его более широко: "Все визуальные элементы приложения (цвета, шрифты и языки) должны настраиваться конечным пользователем".

3. Эта формулировка не является требованием, это архитектура. Когда вы сталкиваетесь с подобным, вам придется копать очень глубоко, чтобы понять, что же думает пользователь.

4. Основное требование, вероятно, выглядит примерно так: "Система предотвращает ввод пользователем недопустимых значений в поля и предупреждает пользователя, если ввод этих значений имеет место".

5. Эта формулировка, по всей вероятности, является жестким требованием.

Решение головоломки с четырьмя точками, приведенной в разделе "Разгадка невероятных головоломок".



notes

Примечания

1

При этом можно утешаться изречением, приписываемым контр-адмиралу д-ру Грэйсу Хопперу: "Легче просить прощения, чем получать разрешение".

2

Это, конечно шутка!

3

Истекающий актив представляет собой некий актив, чья стоимость со временем уменьшается. Примерами могут послужить склад, доверху заполненный бананами, и билет на бейсбол.

4

В эру Интернета, многие люди, кажется, забыли о настоящих библиотеках, в которых имеется штат сотрудников и научные материалы.

5

В оригинале английское слово annoy происходит от старофранцузского *enui* что также означает "наскучить".

Применение функций средств доступа дополняет книгу Мейера "Uniform Access principle" [Mey97b], в которой говорится, что "все службы, обеспечиваемые неким модулем, должны быть доступны за счет универсальной системы обозначений, которая отличается надежностью независимо от того, хранятся ли эти службы в памяти или вычисляются".

7

В вертолете имеется четыре основных органа управления. Рычаг управления циклическим шагом несущего винта находится справа от пилота. При его перемещении вертолет движется в соответствующем направлении. Слева от пилота расположен рычаг управления общим шагом несущего винта. Если потянуть за него, то произойдет увеличение шага на всех лопастях и вертолет начнет подниматься. На конце рычага управления общим шагом расположен дроссель. И наконец, есть две ножных педали, которые изменяют тягу на рулевом винте и способствуют повороту вертолета.

В действительности, это наивно. Лишь при очень большом везении большинство требований из реальной жизни не окажут воздействие на многие функции системы. Тем не менее, в идеале каждое функциональное изменение обязано затрагивать лишь один-единственный модуль.

9

Возьмем нелинейную или хаотическую систему и внесем небольшое изменение в один из входных параметров. Можно получить серьезный и зачастую непредсказуемый результат. Классический пример: взмах крылышек бабочки в Токио может стать началом цепочки событий, приводящих к возникновению смерча в Техасе. Не напоминает ли это явление некоторые известные вам проекты?

10

Если быть педантичным, то существует много способов стрельбы из пулемета в темное время суток, включая стрельбу с закрытыми глазами (поливая свинцом все вокруг). Но это лишь аналогия, и авторам позволительны некоторые вольности.

11

При исследовании абсолютной (в противовес относительной) рабочей характеристики, необходимо придерживаться некоего языка, близкого по характеристикам к целевому языку.

12

Нормальная форма Бэкуса-Наура (BNF) позволяет осуществлять рекурсивное определение контекстно-свободных грамматик. В любой толковой книге по проектированию компиляторов или синтаксическому анализу имеется подробная спецификация формы BNF.

Авторы благодарят Эрика Вота за этот пример.

На самом деле можно приобретать инструментальные средства, поддерживающие только этот способ написания сценариев. Кроме того, можно изучить пакеты с простым исходным текстом (типа Expect), в которых имеются подобные возможности [URL 24]) делается без компиляции. Это существенно упрощает сопровождение в области динамической области приложения.

Для законодателей вполне достаточно значения 3. В законопроекте № 246 Законодательного собрания штата Индиана (1897) была сделана попытка установить, что отныне число «пи» будет равно 3. Во втором чтении законопроект был отложен на неопределенное время, так как некий профессор математики указал, что власть законодателей не распространяется на законы природы.

Для этой цели часто применяется MD5. Великолепное введение в чудесный мир криптографии – книга [Sch95]

17

Все программы становятся унаследованными, как только они написаны.

Генеральная общая лицензия GNU [URL 57] является разновидностью легального вируса, который используется разработчиками программ с открытым текстом для защиты своих (и ваших) прав. Стоит уделить время ее изучению. Она говорит о том, что пользователь может использовать и модифицировать программы с генеральной общей лицензией, но если он распространяет модифицированные программы, то они подлежат соответствующему лицензированию (и маркируются как таковые), а исходный текст должен быть открыт. Это и есть часть вируса – если ваша программа создается на основе лицензированной программы, то она также подлежит лицензированию. Тем не менее, пользователь не ограничен никоим образом при использовании инструментальных средств – право собственности и лицензирование программ, разработанных при помощи указанных средств, находятся на усмотрении пользователя.

В идеальном случае используемая оболочка должна иметь те же клавиатурные привязки, что и редактор. Например, Bash поддерживает клавиатурные привязки редакторов vi и emacs.

Подобным образом разработано ядро Linux. В данном случае имеются разработчики, разбросанные географически, многие из которых работают над одними и теми же фрагментами текста. Опубликован перечень установочных параметров (в данном случае для редактора Emacs), содержащий описание требуемого стиля отступов.

В книге используется английская аббревиатура SCCS (заглавные буквы), которая обозначает системы управления исходным текстом вообще. Помимо этого, существует также особая система управления, обозначаемая sees (строчные буквы), изначально выпущенная фирмой AT&T вместе с Unix System V.

22

Почему "резиновый утенок"? Один из авторов книги, Дэйв Хант, учился в лондонском Империал колледже и много работал совместно с аспирантом, которого звали Грег Паг и которого Д. Хант считает одним из лучших известных ему разработчиков. На протяжении нескольких месяцев Грег носил при себе крохотного желтого резинового утенка, которого он ставил на край монитора во время работы. Прошло некоторое время, пока Дэйв не отважился спросить…

23

В оригинале router обозначает не маршрутизатор ЛВС, а фрезерный станок.

Как насчет создания текста из схемы БД? Существует несколько способов. Если схема содержится в плоском файле (например, операторы create statements), тс синтаксический анализ и генерацию исходного текста можно провести при помощи относительно несложного сценария. В качестве альтернативного способа предлагается следующий: при использовании инструментального средства для создания схемы непосредственно в самой БД необходимо иметь возможность извлечения нужной информации непосредственно из словаря БД. В языке Perl имеются библиотеки, обеспечивающие доступ к большинству основных БД.

Концепция частично основана на ранней работе Дейкстры, Флойда, Хоара, Вирта и др. Более подробная информация о самом языке Eiffel содержится в сети Интернет, см. [URL 10] и [URL 11].

В языках программирования, имеющих своей основой язык C, можно использовать препроцессор или конструкцию с условными операторами с тем, чтобы указанные утверждения не носили обязательного характера. Во многих разработках происходит отключение генерации текста программы для макроса assert при установленном (или сброшенном) флажке этапа компиляции. Можно также поместить текст программы в пределах условного оператора if с постоянным условием, которое многие компиляторы (включая наиболее распространенные Java-системы) отбросят в ходе оптимизации.)

Опасности, возникающие из-за связанности в программе, обсуждаются в разделе "Несвязанность и закон Деметера".

Если n объектов знают друг о друге вес, то при изменении одного-единственного объекта возникает потребность в изменении оставшихся n – 1 объектов.

29

На мирных (читай – глупых) птиц не действовало даже то, что поселенцы забивали их до смерти спортивными битами.

В книге не рассматриваются подробности параллельного программирования; в хорошем учебнике по информатике даются его основы, включая диспетчеризацию, взаимоблокировку, зависание процесса, взаимоисключение/семафоры и т. д.

Более подробная информация обо всех типах диаграмм UML (унифицированного языка моделирования) содержится в книге [FS97].

Несмотря на то, что база данных показана как единое целое, это не так. Программное обеспечение баз данных разделено на несколько процессов и клиентских потоков, но их обработка производится внутренними программами БД и не является частью примера, приведенного в книге.

Она использует статические данные для сохранения текущей позиции в буфере. Статические данные не защищены от параллельного доступа, поэтому они не являются поточно-ориентированными. Помимо этого, программа strtok затирает первый передаваемый параметр, что может привести к весьма неприятным сюрпризам.)

Более подробная информация содержится в описании шаблона Observer в книге [GHJV95].

Представление и контроллер тесно связаны между собой, и в некоторых реализациях MVC они являются единым целым.

36

Тот факт, что самолет пролетает над головой, возможно, не представляет интереса, если только это не сотый самолет за ночь.

В этом случае дело может зайти слишком далеко. Один разработчик переписывал абсолютно все исходные тексты, которые ему передавались, т. к. пользовался собственными соглашениями об именовании.

На деле авторам не хватило реальной памяти для выполнения поразрядной сортировки свыше 7 млн чисел на компьютере с процессором Pentium и 64 Мбайт оперативной памяти во время тестирования алгоритмов, используемых в качестве упражнения к данному разделу. После этого была задействована область подкачки, и время сортировки резко сократилось.

39

Термин "программная интегральная схема", по-видимому, был введен Коксом и Новобилски (1986) в их книге по языку Objective-C под названием "Object-Oriented Programming" [CN91].

Тем не менее существуют иные методики, которые помогают управлять сложностью программ. Две из них – Java beans и AOP – обсуждались в разделе "Ортогональность".

41

Неделя – это долго или нет? На самом деле нет, особенно если рассматривать процессы, в которых менеджмент и исполнители находятся в разных мирах. Менеджмент дает одно представление о том, как все работает, но как только спускаешься в цех. то встречаешься с иной реальностью, для адаптации к которой требуется время.

Существуют некоторые формальные методики, которые пытаются выразить операции алгебраически, но они редко используются на практике. Эти методы требуют, чтобы аналитики разъясняли их значение конечным пользователям.

Подробные спецификации, несомненно, подходят для систем жизнеобеспечения. Очевидно, что эти спецификации составляются для интерфейсов и библиотек, используемых другими пользователями. Если результаты в целом представляют собой набор стандартных вызовов, лучше убедиться в том, что эти вызовы строго определены.

В команде нет разногласий – но это только внешне. Внутри же команды поощряются оживленные серьезные дискуссии. Хорошие разработчики склонны быть страстными, когда речь идет об их работе.

45

В книге "The Rational Unified Process: An Introduction" [Kru98] автор выделяет 27 отдельных ролей в пределах проектной команды!

Например, при записи компакт-диска в формате ISO9660, запускается программа, создающая побитовый образ файловой системы 9660. Стоит ли тянуть до последней минуты, чтобы убедиться, что все нормально?

А что же первое? Проклятый склероз.

Ориентировочно можно принять среднюю величину по отрасли равной $ 100000 на душу; в нее входят заработная плата, социальные выплаты, обучение, оборудование рабочего места, накладные расходы и т. д.

На сайте extreme Programming [URL 45] эта концепция обозначена как "непрерывная интеграция, безжалостное тестирование".

Редактор американского издания требовал изменить это предложение на "Если система выходит из строя… ". Авторы сопротивлялись.

51

В оригинале приводится толкование термина deadline – контрольный срок – в Webster's Collegiate Dictionary: черта, проведенная вокруг тюрьмы (или в ее пределах), за которую заключенный не имеет права выходить под страхом смерти.

Информация подобного рода, как и имя файла, дается тегом RCS Id.

Более подробно модели и представления рассмотрены в разделе "Всего лишь визуальное представление".

Технологии XSL и CSS были разработаны для отделения представления от содержимого.

ACM Member Services, PO BOX 11414, New York, NY 10286, USA. => www.acm.org

1730 Massachusetts Avenue NW, Washington, DC 20036-1992, USA. -> www.computer.org