Содержание

Глава 2

Прагматичный подход

Имеются определенные рекомендации и приемы, применяемые на всех уровнях разработки программного обеспечения, совершенно универсальные процессы и почти аксиоматические идеи. Но такие подходы редко документируются, а вместо этого они зачастую обнаруживаются в виде отрывочных записей дискуссий о проектировании, управлении проектом или программировании. Но ради вашего удобства мы постараемся здесь собрать все эти идеи и процессы вместе.

Первая и, может быть, самая важная тема касается самой сути разработки программного обеспечения и раскрывается в разделе "Сущность качественного проектирования". Из нее следует все остальное. Далее следуют разделы "DRY — пороки дублирования" и "Ортогональность", в которых раскрываются две тесно связанные темы. В первом из них содержится предупреждение не дублировать знания по системам, а во втором — не разделять никакие фрагменты знаний по многим компонентам системы.

По мере увеличения темпов изменений становится все труднее и труднее сохранять приложения в должном состояния. Поэтому в разделе "Обратимость" мы рассмотрим некоторые методики, помогающие ограждать проекты от их изменяющейся среды.

Два последующих раздела также взаимосвязаны. Так, в разделе "Трассирующие пули" речь пойдет о стиле разработки, позволяющем одновременно собирать требования, проверять проектные решения и реализовывать код. И это единственный способ идти в ногу с современной жизнью. А в разделе "Прототипы и памятные записки" будет показано, каким образом прототипирование применяется для проверки архитектур, алгоритмов, интерфейсов и идей. В современном мире крайне важно проверять идеи и получать ответную реакцию, прежде чем безоговорочно принимать их.

По мере созревания вычислительной техники разработчики во все большем количестве создают языки высокого уровня. И хотя еще не изобретен компилятор, способный принимать команду "сделай это вот так", в разделе "Предметно-ориентированные языки" представлены более скромные рекомендации, которые вы можете реализовать самостоятельно.

Наконец, все мы работаем в условиях ограниченного времени и ресурсов. Нехватку таких ресурсов можно перенести легче (и принести большее удовлетворение начальству или клиентам), если постараться выяснить, как долго они будут затребованы. Именно об этом и пойдет речь в разделе "Оценивание".

Упомянутые выше принципы следует непременно иметь в виду во время разработки, чтобы писать более совершенный, быстродействующий и устойчивый код. Вы можете даже сделать его более удобочитаемым.

Тема 8 Сущность качественного проектирования

В мире полно умников и знатоков проектирования программного обеспечения, жаждущих передать свою мудрость, приобретенную тяжкими трудами. Для этого существуют особые сокращения, списки (почему-то обычно из пяти пунктов), шаблоны, диаграммы, видеоматериалы, беседы, а возможно, и увлекательные сериалы (Интернет есть Интернет), поясняющие закон Деметры с помощью интерпретирующего танца.

И мы, авторы этой книги, грешим этим. Но нам хотелось бы внести поправки в пояснение того, что лишь стало для нас очевидным. Прежде всего сделаем заявление общего характера.

Совет 14 Удачное проектное решение легче изменить, чем неудачное

Вещь считается удачно спроектированной, если она приспосабливается к тем, кто ею пользуется. Для кода это означает, что он должен приспосабливаться к изменениям. Поэтому мы верим в принцип *легкости изменения* (Easier to Change — ETC). Вот и все.

Насколько мы можем судить, всякий принцип проектирования является частным случаем принципа ETC. Чем, например, хорошо уменьшение степени связывания? Оно хорошо тем, что разделяет обязанности, чтобы их легче было изменить. А это и есть принцип ETC. В чем польза принципа единственной ответственности? Она состоит в том, что изменения в требованиях зеркально отображаются в изменениях лишь в одном модуле. И это соответствует принципу ETC. Почему так важно именование? А потому, что удачные имена делают исходный код более удобочитаемым, а ведь его приходится читать, чтобы внести в него изменения. И в этом случае соблюдается принцип ETC!

Принцип ETC — это ценность, а не правило

Ценности — это сущности, помогающие принять решение сделать то или другое. Что касается осмысления программного обеспечения, то ETC является

руководящим принципом, помогающим выбрать правильный путь. Как и все остальные ценности, этот принцип должен плавно следовать за здравой мыслью, мягко подталкивая в правильном направлении.

Но как этого добиться? Как показывает наш опыт, для этого требуется первоначальное сознательное подкрепление. Вам, возможно, придется потратить около недели, сознательно задавая себе вопрос: "Облегчило или же затруднило изменение всей системы в целом то, что я только что сделал?" Делайте это, когда сохраняете файл, пишете тест или исправляете ошибку в программе.

В принципе ETC присутствует неявное предположение. Он предполагает, что человек способен выбрать из многих путей именно тот, который будет легче сменить впоследствии. Зачастую здравый смысл позволяет выбрать правильный путь и дает возможность сделать обоснованное предположение. Но иногда на это нет и намека. Что ж, бывает и так. Мы считаем, что в подобных случаях можно сделать две вещи.

Во-первых, если вы не знаете, какую конечную форму примут вносимые вами изменения, то всегда можете вернуться к пути "легкости изменений": попытаться сделать заменяемым фрагмент кода, который вы пишете. В таком случае, что бы ни произошло впоследствии, этот фрагмент кода не станет неодолимой преградой на вашем пути. Такая мера кажется крайней, но на самом деле вы должны так или иначе прибегать к ней постоянно. Это всего лишь забота о сохранении кода не связанным и согласованным.

Во-вторых, рассматривайте это как способ развития инстинктов. Запишите данную ситуацию в своем журнале технического учета, упомянув выбранные вами варианты и некоторые предположения об изменениях. Оставьте метку в исходном коде. В дальнейшем, когда этот фрагмент кода придется изменить, вы сможете оглянуться назад и предоставить отчет самому себе. Это может помочь, когда вы в очередной раз окажетесь на аналогичной развилке.

В остальных разделах этой главы рассматриваются конкретные идеи по поводу проектирования. Но все они вызваны описанным здесь одним и тем же принципом легкости изменения.

Другие разделы, связанные с данной темой

- **Тема 9**. DRY — пороки дублирования.
- **Тема 10**. Ортогональность.
- **Тема 11**. Обратимость.
- **Тема 14**. Предметно-ориентированные языки.
- **Тема 28**. Развязывание, **глава 5** "Гибкость или ломкость".
- **Тема 30**. Преобразовательное программирование, **глава 5** "Гибкость или ломкость".
- **Тема 31**. Налог на наследование, **глава 5** "Гибкость или ломкость".

Задачи

- Обдумайте принцип проектирования, которым пользуетесь регулярно. Направлен ли он на то, чтобы сделать что-то легко изменяемым?
- Подумайте также о языках и парадигмах программирования: объектно-ориентированного, функционального и т.п. Какие главные положительные, отрицательные или те и другие стороны они имеют для написания кода в соответствии с принципом ETC?
- Что вы можете сделать при программировании для того, чтобы исключить отрицательные и выделить положительные стороны[1]?
- Во многих текстовых редакторах имеется (встроенная или через расширения) поддержка команд, выполняемых при сохранении файла. Настраивайте текстовый редактор на вывод сообщения `"ETC?"` всякий раз[2], когда сохраняете изменения в файле. Пользуйтесь такой возможностью в качестве напоминания о необходимости обдумать написанный вами код. Насколько легко его изменить?

Тема 9 DRY — пороки дублирования

Капитан Джеймс Тиберий Кирк[3] предпочитал снабжать компьютер двумя противоречащими друг другу фрагментами знаний, чтобы нейтрализовать враждебный искусственный интеллект. К сожалению, тот же самый принцип может очень легко погубить *ваш* код.

Программисты обычно собирают, организуют, хранят и используют знания. Они документируют знания в спецификациях, возрождают их в выполняемом коде и пользуются ими для проверок во время тестирования. К сожалению, знания непостоянны. Они изменяются, и зачастую очень быстро. Так, ваше представление о каком-нибудь требовании может измениться после совещания с клиентом. Стоит правительству изменить какой-нибудь нормативный акт, и соответствующая бизнес-логика устаревает. Тесты могут показать, что выбранный алгоритм неработоспособен. Все это непостоянство означает, что нам приходится бо́льшую часть времени работать в режиме сопровождения, реорганизации и преобразования знаний в своих системах.

---

[1] Перефразируя старую песню *Accentuate The Positive* (Делайте акцент на положительном) Джонни Мерсера (Johnny Mercer) — американского исполнителя, популярного в 1940-х годах.

[2] Или хотя бы каждый десятый раз, чтобы оставаться в здравом уме…

[3] Персонаж научно-фантастического телевизионного сериала *Звёздный путь: Оригинальный сериал* (Star Trek: The Original Series). — *Примеч. пер.*

Многие полагают, что сопровождение приложения начинается после его выпуска и означает устранение программных ошибок и совершенствование функциональных средств. Мы считаем, что они не правы. Программисты постоянно работают в режиме сопровождения, поскольку их представления изменяются день за днем, когда поступают новые требования, а уже имеющиеся требования развиваются по мере сосредоточения усилий на проекте. Может измениться и сама среда. Но, независимо от конкретной причины, сопровождение является не каким-то отдельным видом деятельности, а обыкновенной стадией всего процесса разработки.

Когда выполняется сопровождение, приходится искать и изменять представления вещей, т.е. те крупицы знаний, которые вложены в приложение. Но дело в том, что знания очень легко продублировать в спецификациях, процессах и разрабатываемых программах. И делая это, разработчики навлекают сущий кошмар сопровождения, который начинается еще до выпуска приложения. Мы считаем, что единственный способ надежно разрабатывать программное обеспечение и упростить понимание и сопровождение разработок — следовать принципу "не повторяться" (DRY):

> *У каждого фрагмента знаний должно быть единственное, недвусмысленное, непререкаемое представление в системе.*

А почему этот принцип называется DRY? А вот почему:

Совет 15 DRY — Don't Repeat Yourself, т.е. "не повторяйся"

Альтернатива состоит в присутствии чего-то одного в двух или более местах. Если в таком случае изменить что-то одно, то нужно не забыть изменить и все остальное, а иначе программа будет загнана противоречием в угол, как компьютеры пришельцев. И вопрос не в том, чтобы вспомнить об изменениях, а в том, когда они забудутся.

Принцип DRY будет еще не раз упоминаться на страницах этой книги, и зачастую в таком контексте, который не имеет ничего общего с программированием. Мы считаем, что это одно из самых важных инструментальных средств в арсенале программиста-прагматика. И в этом разделе мы вкратце изложим трудности, возникающие в связи с дублированием, а также предложим общие стратегии их преодоления.

Принцип DRY не только для кодирования

Прежде всего устраним некоторое недоразумение. В первом издании этой книги мы поступили неверно, объяснив лишь, что мы подразумевали под выражением "не повторяться". И многие посчитали, что принцип DRY имеет отношение только к коду, думая, что он означает "не выполнять копирование и вставку

строк исходного кода". Но это лишь *часть* принципа DRY, причем мелкая и самая простая. Принцип DRY имеет отношение к дублированию *знаний*, а также *намерений*, т.е. к выражению одного и того же в двух разных местах, кроме того, возможно, разными способами.

В качестве пробного камня попробуйте ответить на следующие вопросы: если должно быть изменено какое-нибудь свойство кода, то не придется ли вносить изменения во многих местах и в самых разных форматах? Не придется ли изменять исходный код и документацию, схему базы данных и хранящуюся в ней структуру и т.д.? Если это именно так, то такой код не соответствует принципу DRY. Рассмотрим некоторые типичные примеры дублирования.

Дублирование в исходном коде

Каким бы банальным это ни показалось, но дублирование кода — довольно частое явление. Ниже приведен характерный тому пример.

```
def print_balance(account)
    printf "Debits: %10.2f\n", account.debits
    printf "Credits: %10.2f\n", account.credits
    if account.fees < 0
        printf "Fees: %10.2f-\n", -account.fees
    else
        printf "Fees: %10.2f\n", account.fees
    end
    printf " ———-\n"
    if account.balance < 0
        printf "Balance: %10.2f-\n", -account.balance
    else
        printf "Balance: %10.2f\n", account.balance
    end
end
```

Пренебрежем пока что следствиями типичной для новичков ошибки, работая с денежными суммами в числовом формате с плавающей точкой. Вместо этого попробуем выявить дублирование в приведенном выше фрагменте кода. (Мы обнаружили три таких места, но вы можете найти и больше.)

Что же мы нашли? А вот что.

Прежде всего, в рассматриваемом здесь коде явно проявляется дублирование обработки отрицательных чисел методом копирования и вставки. Этот недостаток можно устранить, введя еще одну функцию, как показано ниже.

```
def format_amount(value)
    result = sprintf("%10.2f", value.abs)
    if value < 0
        result + "-"
    else
        result + " "
    end
end
```