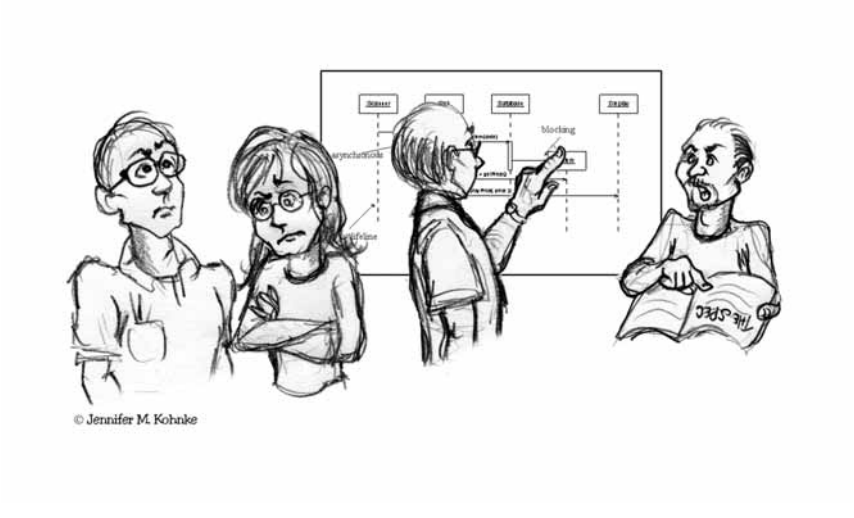
Содержание

Глава 7

Введение в гибкое проектирование



После критического анализа жизненного цикла разработки ПО, как я его понимаю, я сделал вывод, что единственной документацией, которая действительно в состоянии удовлетворять критерию инженерного проектирования, являются листинги исходного кода.

Джек Ривз

Джек Ривз опубликовал в журнале *C++ Journal* конструктивную статью под названием “What is Software Design?” (“Что такое программный проект?”)[2]. В этой статье Ривз доказывает, что проект программной системы документируется главным образом ее исходным кодом. Диаграммы, представляющие исходный код, являются дополнением к проекту, но не самим проектом. Как оказалось впоследствии, статья Джека стала предвестником гибкой разработки.

Далее мы часто будем говорить о “проекте”. Вы не должны считать, что речь идет о наборе диаграмм UML, отдельном от кода. Набор диаграмм UML может представлять части проекта, но это не *сам* проект. Проект программной системы — абстрактная концепция. Он имеет отношение к общей форме и структуре программы, а также детальной форме и структуре каждого модуля, класса и метода. Проект может быть представлен с помощью множества разных носителей, но его окончательным воплощением является исходный код. В конечном счете, исходный код и есть сам проект.

Что не так с программным обеспечением?

Если вам везет, то вы начинаете работу, имея ясное представление о том, какой должна быть окончательная система. Проект системы — это живой образ в вашем уме. При еще большем везении ясность проекта сохранится до первого выпуска.

Затем что-то идет не так. Программа начинает разлагаться, словно кусок испорченного мяса. С течением времени разложение ширится и усиливается. В коде накапливаются опасные гнойные язвы и нарывы, делая его все труднее и труднее в сопровождении. В конце концов, сами усилия, требуемые для внесения даже простейших изменений, становятся настолько обременительными, что разработчики и ведущие менеджеры молят о перепроектировании.

Такое перепроектирование редко бывает успешным. Хотя проектировщики начинают с самыми благими намерениями, они обнаруживают, что имеют дело с крайне неустойчивой целью. Старая система продолжает развиваться и изменяться, а новый проект должен следовать с ней наравне. Пороки в новом проекте накапливаются еще задолго до первого выпуска.

[2] [1] Это замечательная работа. Мы настоятельно рекомендуем прочитать ее. Она включена в приложение Г.

Признаки плохого проекта — особенности разлагающегося программного обеспечения

Вы узнаете, что ПО разлагается, когда начнете наблюдать любые из перечисленных ниже признаков.

1. *Жесткость* — систему трудно изменять, потому что каждое изменение вызывает множество дополнительных изменений в других частях системы.
2. *Хрупкость* — изменения приводят к разрушению системы в местах, которые концептуально не связаны с местом, где вносились изменения.
3. *Монолитность* — систему трудно разделить на компоненты, которые могли бы использоваться повторно в других системах.
4. *Вязкость* — делать что-то правильно труднее, чем делать что-то неправильно.
5. *Неоправданная сложность* — проект содержит инфраструктуру, не приносящую непосредственной пользы.
6. *Неоправданная повторяемость* — проект содержит повторяющиеся структуры, которые можно было бы унифицировать в виде единственной абстракции.
7. *Непрозрачность* — проект трудно читать и понимать; он не выражает свой замысел достаточно ясно.

Жесткость

Жесткость — это склонность программы быть сложной в изменении даже простыми способами. Проект является жестким, если одиночное изменение вызывает каскад последующих изменений в зависимых модулях. Чем больше модулей должно быть изменено, тем более жестким считается проект.

Большинству разработчиков приходилось, так или иначе, сталкиваться с этой ситуацией. Их просят внести на первый взгляд простое изменение. Они тщательно изучают это изменение и делают обоснованную оценку требуемой работы. Но позже во время реализации изменения они обнаруживают непредвиденные последствия от его внесения. Им приходится отслеживать изменение, перелопачивая огромные порции кода и модифицируя намного больше модулей, чем предполагалось вначале. В итоге изменения вносятся гораздо дольше, чем было оценено первоначально. Когда их спрашивают, почему подсчитанная ими оценка оказалась настолько далекой от реальности, они повторяют традиционную жалобу разработчиков ПО: “Это оказалось намного сложнее, чем мы думали!”.

Хрупкость

Хрупкость — это склонность программы разрушаться во многих местах после того, как внесено единственное изменение. Зачастую новые проблемы возникают в областях, не имеющих концептуальной связи с областью, которая была изменена. Исправление этих проблем приводит к возникновению еще большего числа проблем, и команда разработчиков начинает походить на пса, гоняющегося за собственным хвостом.

С ростом хрупкости модулей вероятность того, что изменение приведет к непредвиденным проблемам, приближается к 100%. Хоть подобное кажется абсурдным, такие модули отнюдь не редкость. Это те модули, которые постоянно нуждаются в исправлении — модули, никогда не покидающие список ошибок; модули, которые, как известно разработчикам, должны быть перепроектированы (но никто не желает сталкиваться с угрозой их перепроектирования); модули, которые становятся тем *хуже*, чем больше их исправляют.

Монолитность

Проект считается монолитным, когда он содержит части, которые могли бы быть полезными в других системах, но усилия и риски, связанные с их отделением от исходной системы, слишком велики. Это печальный, но очень распространенный случай.

Вязкость

Вязкость встречается в двух формах: вязкость ПО и вязкость среды.

Столкнувшись с необходимостью внести изменение, разработчики обычно находят более одного способа сделать это. Некоторые способы предохраняют проект, а некоторые — нет (т.е. относятся к хакерским приемам). Когда методы, предохраняющие проект, использовать труднее, чем хакерские приемы, то вязкость проекта высока. Делать что-то неправильно легко, но трудно сделать что-нибудь правильно. Мы хотим проектировать ПО так, чтобы можно было легко вносить изменения, предохраняющие проект.

Вязкость среды возникает, когда среда разработки является медленной и неэффективной. Например, если время компиляции очень велико, то у разработчиков возникнет соблазн вносить такие изменения, которые не приводят к массивной перекомпиляции, даже с учетом того, что они не предохраняют проект. Если фиксация всего нескольких файлов в системе управления исходным кодом занимает часы, то разработчики будут склонны вносить изменения, которые требуют настолько мало фиксаций, насколько это возможно, не обращая внимания, предохраняется проект или нет.

В обоих случаях вязкость связана с трудностью предохранения проекта ПО. Мы хотим создавать системы и строить среды, которые делают легким предохранение проекта.

Неоправданная сложность

Проект характеризуется неоправданной сложностью, если он содержит элементы, которые не приносят пользы в текущий момент. Это часто происходит, когда разработчики предвосхищают изменения в требованиях и помещают в ПО средства, призванные справиться с такими потенциальными изменениями. На первых порах это может выглядеть неплохо. В конце концов, подготовка к будущим изменениям должна сохранить код гибким и предотвратить кошмар доработок в будущем.

К сожалению, результат часто оказывается противоположным. Из-за подготовки к слишком многим непредвиденным обстоятельствам проект засоряется конструкциями, которые никогда не задействуются. Некоторые из приготовлений могут окупиться, но очень многие — нет. А тем временем проект несет в себе бремя этих неиспользуемых проектных элементов. В итоге ПО становится сложным и трудным для понимания.

Неоправданная повторяемость

Вырезание и вставка могут быть удобными операциями при редактировании текста, но пагубными при редактировании кода. Слишком часто программные системы строятся на основе десятков или сотен повторяющихся фрагментов кода. Вот как это происходит.

Ральфу необходимо написать код, который хрюкотает зелюков[3]. Он просматривает другие части кода, подозревая, что зелюков уже хотя бы раз хрюкотали, и находит подходящий участок кода. Затем Ральф вырезает и вставляет этот код в свой модуль и вносит подходящие модификации.

Но Ральф не знал, что скопированный им код был помещен туда Тоддом, который в свою очередь вырезал его из модуля, написанного Лили. Она была первой, кто хрюкотал зелюков, но она осознала, что хрюкотание зелюков очень похоже на хрюкотание мюмзиков. Лили нашла где-то код, который хрюкотал мюмзиков, вырезала его и вставила в свой модуль, после чего должным образом модифицировала.

Когда тот же самый код встречается многократно в слегка отличающихся формах, разработчикам не хватает абстракции. Отыскание всех повторяющихся фрагментов и их устранение посредством подходящей абстракции, возможно, не находится в первых пунктах списка приоритетов, но может иметь большое значение в вопросах облечения понимания и сопровождения системы.

При наличии избыточного кода в системе работа по ее изменению становится трудной. Ошибки, обнаруженные в повторяющемся блоке, должны быть исправлены во всех его вхождениях. Однако поскольку вхождения несколько отличаются друг от друга, исправления не всегда будет тем же самым.

---

[3] Неведомая функция программы. Взято из стихотворения “Бармаглот” Льюиса Кэрролла (“Алиса в Зазеркалье”) в переводе Д. Орловской.

Непрозрачность

Непрозрачность — это склонность модуля быть сложным для понимания. Код может быть написан в ясной и выразительной или в темной и запутанной манере. Код, который развивается с течением времени, имеет тенденцию становиться все более и более непрозрачным. Чтобы свести непрозрачность к минимуму, требуются постоянные усилия по сохранению кода ясным и выразительным.

Когда разработчики впервые пишут модуль, код может казаться им ясным. Причина в том, что они ушли в него с головой и понимают на глубинном уровне. Позже, после утраты тесной связи, разработчики возвращаются к этому модулю и удивляются, как они могли написать что-то настолько ужасное. Чтобы предотвратить подобную ситуацию, разработчики должны ставить себя на место читателей кода и прикладывать согласованные усилия по рефакторингу своего кода, делая его понятным читателям. Они также должны давать код на пересмотр другим.

Что побуждает программное обеспечение разлагаться?

В негибких средах проекты деградируют, потому что требования изменяются такими путями, которые не были предугаданы в их первоначальных версиях. Часто изменения необходимо вносить быстро, и это может делаться разработчиками, которые не знакомы с философией исходного проекта. Следовательно, хотя изменение в проекте работает, оно тем или иным образом нарушает первоначальный проект. Мало помалу, по мере того, как изменения продолжают вноситься, такие нарушения накапливаются, и проект начинает разлагаться.

Тем не менее, мы не можем винить в деградации проекта корректировки требований. Как разработчики ПО, мы очень хорошо знаем, что требования меняются. На самом деле большинство из нас понимает, что требования являются самыми непостоянными элементами в проекте. Если проекты разрушаются под напором непрекращающегося потока меняющихся требований, то виноваты в этом наши проекты и методики. Мы должны как-то отыскать способ сделать свои проекты устойчивыми к таким изменениям и задействовать методики, которые защищают их от разложения.

Гибкие команды не позволяют программному обеспечению разлагаться

Гибкая команда буквально расцветает при появлении изменения. Команда не тратит много усилий заранее, поэтому не переживает по поводу устаревания первоначального проекта. Наоборот, разработчики в команде сохраняют проект системы как можно более чистым и простым, подстраховывая его множеством модульных и приемочных тестов. В результате проект делается податливым и легким в изменении. Команда использует преимущества этой гибкости для того, чтобы постоянно улучшать проект, так что каждая итерация заканчивается системой, проект которой максимально соответствует требованиям данной итерации.

Программа Copy

Проиллюстрировать приведенные выше соображения поможет наблюдение за разложением проекта. Предположим, что рано утром в понедельник к вам является начальник и просит написать программу, которая копирует символы, введенные с клавиатуры, на принтер. Быстро прокрутив в голове возможные решения, вы приходите к выводу, что программа будет содержать менее десяти строк кода. Проектирование и написание кода займет не более часа. Но с учетом групповых совещаний, занятий по обеспечению качества, ежедневных встреч по поводу хода работ и трех критических ситуаций на местах на завершение программы потребуется около недели, да и то, если трудиться сверхурочно. Однако вы неизменно умножаете свои оценки на три.

"Три недели" — сообщаете вы начальнику. Он хмыкает и уходит, оставляя вас наедине с задачей.

Первоначальный проект

Перед началом совещания, посвященного анализу процесса, у вас выдалась свободная минутка, поэтому вы решили составить проект программы. С применением структурного проектирования вы получаете блок-схему, показанную на рис. 7.1.

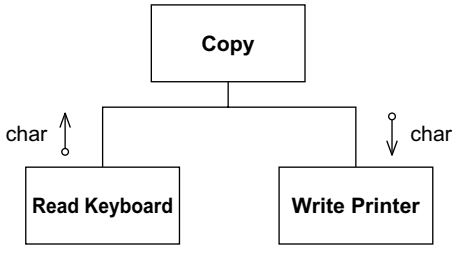


Рис. 7.1. *Блок-схема программы* `Copy`

В программе имеются три модуля, или подпрограммы. Модуль `Copy` (Копирование) обращается к двум другим модулям. Программа `Copy` получает символы от модуля `Read Keyboard` (Чтение с клавиатуры) и направляет их модулю `Write Printer` (Вывод на принтер).

Вы смотрите на свой проект и видите, что он хорош. С улыбкой на лице вы покидаете свой офис и отправляетесь на совещание. По крайней мере, там можно будет хоть немного вздремнуть.

Во вторник вы приходите в офис пораньше, чтобы завершить программу `Copy`. К несчастью, прошлой ночью дала о себе знать одна из критических ситуаций на местах, поэтому вам нужно бежать в лабораторию, чтобы помочь в исправлении проблемы. Во время обеденного перерыва, на который вам удалось уйти в 3 часа дня, вы набираете код программы `Copy` (листинг 7.1).

Листинг 7.1. Программа Copy

```
void Copy()
{
  int c;
  while ((c=RdKbd()) != EOF)
    WrtPrt(c);
}
```

Едва успев сохранить набранный код, вы понимаете, что уже опоздали на занятие по обеспечению качества. Вы знаете, что оно очень важно; там собирались говорить о значимости отсутствия дефектов. Быстро проглотив бутерброд и запив его колой, вы со всех ног мчитесь на занятие.

В среду вы снова приходите рано утром, и на этот раз кажется, ничего плохого не произошло. Вы загружаете исходный код программы `Copy` и приступаете к его компиляции. Гляди-ка, сразу компилируется без ошибок! Это хорошо еще и потому, что начальник вызывает на внеплановое совещание по поводу экономии тонера для лазерных принтеров.

В четверг после того, как вы потратили четыре часа на телефонный разговор с техником по обслуживанию из Роки-Маунт (Северная Калифорния), запуская с ним команды удаленной отладки и регистрации в журнале ошибок для одного из самых скрытых компонентов системы, вы хватаете шоколадку и начинаете тестировать программу `Copy`. Работает, да еще с первого раза! Это хорошо, учитывая, что ваш практикант только что удалил главный каталог с исходным кодом на сервере, и вам придется найти магнитные ленты с последней резервной копией и провести восстановление. Конечно же, последняя полная резервная копия создавалась три месяца назад, так что понадобится восстановить поверх нее еще девяносто четыре инкрементных копии.

На пятницу совершенно ничего не запланировано. И это тоже хорошо, т.к. загрузка программы `Copy` в систему управления исходным кодом заняла целый день.

Разумеется, программа имеет громкий успех и развертывается по всей компании. Ваша репутация как первоклассного программиста в очередной раз подтверждена, и вы наслаждаетесь славой своих достижений. Если повезет, то в этом году вы действительно сможете написать целых тридцать строк кода!

Тем временем требования меняются

Через несколько месяцев заходит ваш начальник и говорит, что временами хотелось бы, чтобы программа `Copy` была способна читать с перфоленты. Вы скрежещете зубами и сердито вращаете глазами. Вам интересно, почему люди всегда изменяют требования. Ведь ваша программа не была рассчитана на чтение с перфоленты! Вы предупреждаете начальника, что изменения такого рода уничтожат элегантность проекта. Тем не менее, начальник непреклонен. Он утверждает, что пользователям время от времени действительно нужно читать символы с перфоленты.

Итак, вздохнув, вы приступаете к планированию модификаций. Вам хотелось бы добавить в функцию `Copy()` булевский аргумент. Если он равен `true`, то будет производиться чтение с перфоленты, а если `false` — то с клавиатуры, как и раньше. К сожалению, в настоящее время функцию `Copy()` используют настолько много других программ, что изменять ее интерфейс нельзя. Изменение интерфейса вызовет многие недели повторной компиляции и тестирования. Да системные тестировщики в одиночку линчевали бы вас, не говоря уже о тех семи парнях из группы управления конфигурацией. А сотрудники службы контроля за процессом получили бы потрясающее удовольствие, применяя все виды пересмотра кода к каждому модулю, который вызывает функцию `Copy()`!

Нет, изменение интерфейса исключено. Но как тогда указать программе `Copy`, что она должна читать с перфоленты? Ну конечно, вы будете использовать глобальную переменную! А еще вы будете применять самое лучшее и полезное средство языкового семейства C — операцию `?:`! Результат показан в листинге 7.2.

Листинг 7.2. Первая модификация программы Copy

```
bool ptFlag = false;
// не забудьте сбросить этот флаг
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(c);
}
```

Вызывающие `Copy()` модули, которым необходимо выполнять чтение с перфоленты, должны сначала установить переменную `ptFlag` в `true`. Затем они могут вызывать функцию `Copy()` и она будет благополучно читать с перфоленты. После завершения `Copy()` вызывающий модуль должен сбросить `ptFlag`, иначе следующий вызывающий модуль может ошибочно производить чтение с перфоленты, а не с клавиатуры. Чтобы напомнить программистам об их обязанности сбросить этот флаг, вы предусмотрели соответствующий комментарий в коде.

Вы снова выставляете свою программу на суд публики. Она оказывается даже более успешной, чем ранее, и полчища нетерпеливых программистов ждут возможности попользоваться ею. Жизнь удалась.

Дайте им еще чуть-чуть...

Несколько недель спустя ваш начальник (да, он по-прежнему ваш начальник, несмотря на три корпоративных реорганизации за последние месяцы) сообщает вам о том, что заказчики хотели бы, чтобы программа `Copy` иногда выводила на перфоленту.

Заказчики! Они всегда разрушают ваши проекты. *Писать программы было бы намного легче, если бы они не предназначались для заказчиков.*

Вы говорите начальнику, что такие непрекращающиеся изменения оказывают крайне негативное влияние на элегантность проекта. Вы предупреждаете его, что если изменения продолжат поступать в таком бешеном темпе, то до конца года программу будет невозможно сопровождать. Начальник понимающе кивает и затем изрекает, что изменение должно быть внесено в любом случае.

Это изменение проекта похоже на предыдущее изменение. Понадобится лишь еще одна глобальная переменная и еще одна операция `?:`! В листинге 7.3 приведен результат приложенных усилий.

Листинг 7.3. Вторая модификация программы Copy

```
bool ptFlag = false;
bool punchFlag = false;
// не забудьте сбросить эти флаги
void Copy()
{
  int c;
  while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
    punchFlag ? WrtPunch(c) : WrtPrt(c);
}
```

Вы особенно горды тем, что не забыли изменить комментарий. Однако вас беспокоит, что структура программы начинает разваливаться. Любые дальнейшие изменения устройства ввода определенно заставят вас полностью реструктуризировать условие в цикле `while`. Может быть, самое время сдуть пыль со своего резюме...

Ожидайте изменений

Мы оставляем вам право самим решить, сколько в приведенном рассказе сатирических преувеличений. Суть истории в том, чтобы показать, что проект программы может быстро деградировать при появлении изменения. Исходный проект программы `Copy` был простым и элегантным. После всего лишь двух изменений он начал демонстрировать признаки жесткости, хрупкости, монолитности, сложности, избыточности и непрозрачности. Данная тенденция, несомненно, будет продолжаться, и в программе наступит беспорядок.

Можно расслабиться и винить в этом изменения. Можно посетовать на то, что программа была хорошо спроектирована для исходной спецификации, а последующие изменения, внесенные в спецификацию, вызвали деградацию проекта. Тем не менее, в таком случае игнорируется один из самых примечательных фактов при разработке ПО: *требования изменяются всегда*!

Помните, что наиболее изменчивой частью в большинстве программных проектов являются требования. Требования находятся в состоянии постоянного движения. Это факт, который мы, разработчики, обязаны признать!

Мы живем в мире изменяющих требований, и наша работа заключается в том, чтобы создаваемое ПО могло пережить эти изменения. Если программный проект деградирует из-за изменения требований, то ни о какой гибкости речи быть не может.

Гибкий проект примера программы Copy

Гибкая разработка могла бы начинаться с того же самого кода, который показан в листинге 7.1[4]. Когда начальник просит гибких разработчиков сделать так, чтобы программа читала с перфоленты, они реагируют модификацией проекта, делающей его упругим к этому виду изменений. Результат мог бы выглядеть подобным представленному в листинге 7.4.

Листинг 7.4. Гибкая версия программы Copy

```
class Reader
{
  public:
    virtual int read() = 0;
};
class KeyboardReader : public Reader
{
  public:
    virtual int read() {return RdKbd();}
};
KeyboardReader GdefaultReader;
void Copy(Reader& reader = GdefaultReader)
{
  int c;
  while ((c=reader.read()) != EOF)
    WrtPrt(c);
}
```

Вместо попытки исправить проект, чтобы заработало новое требование, команда пользуется возможностью улучшить проект, сделав его упругим к такому виду изменений в будущем. С этого момента, когда бы начальник ни попросил добавить новый тип устройства ввода, команда будет способна отреагировать так, чтобы не вызвать деградацию программы `Copy`.

Команда следовала принципу открытости/закрытости (OCP), который обсуждается в главе 9. Данный принцип предписывает проектировать модули таким образом, чтобы их можно было расширять без модификации. Именно это

[4] В действительности весьма возможно, что методика разработки через тестирование обеспечила бы проекту достаточную гибкость, которая позволила бы выдержать запросы начальника вообще без модификации. Однако в рассматриваемом примере мы это игнорируем.

команда и сделала. Каждое новое устройство ввода, требуемое начальником, может быть предоставлено без модификации программы `Copy`.

Тем не менее, обратите внимание, что при первоначальном проектировании модуля команда вовсе не пыталась предвидеть, что программа будет изменяться. Взамен обработчики написали ее насколько просто, насколько смогли. И только когда требования, в конце концов, поменялись, они изменили проект модуля, сделав его упругим к изменениям такого рода.

Можно было бы утверждать, что команда сделала только половину работы. Наряду с тем, что разработчики защитили себя от разнообразных устройств ввода, они также могли бы защититься и от различных устройств вывода. Однако команда действительно не имела никакого представления о том, что устройства вывода когда-либо изменятся. Добавление дополнительной защиты было бы работой, не служащей каким-то текущим целям. Ясно, что если такая защита понадобится, то ее легко будет добавить позже. Следовательно, добавлять ее сейчас совершенно нет причин.

Как гибкие разработчики узнали, что делать?

Гибкие разработчики в рассмотренном выше примере построили абстрактный класс, чтобы защититься от изменений устройства ввода. Как они узнали, что нужно поступить именно так? Это должно было делаться согласно одному из фундаментальных принципов объектно-ориентированного проектирования.

Первоначальный проект программы `Copy` не является гибким из-за *направления* его зависимостей. Еще раз взгляните на рис. 7.1. Обратите внимание, что модуль `Copy` зависит напрямую от модулей `Read Keyboard` и `Write Printer`. В этом приложении `Copy` — модуль верхнего уровня. Он устанавливает политику приложения. Ему известно, как копировать символы. К сожалению, он также сделан зависимым от низкоуровневых деталей клавиатуры и принтера. Таким образом, изменение низкоуровневых деталей затрагивает высокоуровневую политику.

Как только негибкость проявилась, гибкие разработчики поняли, что зависимость модуля `Copy` от устройства ввода необходимо инвертировать (глава 11), чтобы `Copy` больше не зависел от устройства ввода. Для создания желаемой инверсии они задействовали паттерн Стратегия (Strategy), описанный в главе 14.

Словом, гибкие разработчики знали, что делать, по перечисленным ниже причинам.

1. Они обнаружили проблему за счет следования гибким методикам.
2. Они диагностировали проблему путем применения принципов гибкого проектирования.
3. Они устранили проблему с использованием подходящего паттерна.

Взаимодействие между этими тремя аспектами разработки ПО и *является* процессом проектирования.

Поддержание проекта в как можно лучшем виде

Гибкие разработчики отдают все силы на поддержание проекта в максимально адекватном и чистом состоянии. Это не какая-то бессистемная или временная обязанность. Гибкие разработчики вовсе не "приводят в порядок" проект раз в несколько недель. Они сохраняют ПО как можно более чистым, простым и выразительным ежедневно, ежечасно и даже ежеминутно. Они никогда не говорят: "Мы исправим это позже". Они вообще не позволяют начаться разложению.

Гибкие разработчики относятся к проектированию ПО точно так же, как хирурги относятся к предстоящей процедуре стерилизации. Процедура стерилизации — это то, что делает хирургию *возможной*. Без нее риск заражения был бы недопустимо высоким. Гибкие разработчики испытывают то же чувство в отношении своих проектов. Риск допустить появление даже мельчайшего очага разложения слишком велик, чтобы с ним можно было мириться.

Проект должен оставаться чистым, а поскольку исходный код является наиболее важным представлением проекта, то и он должен оставаться чистым. Профессионализм требует, чтобы мы, разработчики ПО, не позволяли коду разлагаться.

Заключение

Итак, что такое гибкое проектирование? Гибкое проектирование — это процесс, а не разовое мероприятие. Это непрерывное применение принципов, паттернов и методик для улучшения структуры и читабельности ПО. Это посвящение себя поддержанию проекта системы в как можно более простом, чистом и выразительном состоянии на долговременной основе.

В последующих главах мы займемся исследованиями принципов и паттернов проектирования ПО. Во время их чтения помните, что гибкие разработчики не применяют эти принципы и паттерны к крупному, учитывающему все аспекты проекту. Наоборот, они применяют их от итерации к итерации в попытке сохранить чистоту кода и воплощенного в нем проекта.

Библиография

1. Reeves, Jack. What Is Software Design? *C++ Journal*, Vol. 2, No. 2. 1992. Доступно по адресу `http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm`.