Глава 1
Объектно-ориентированное проектирование

Наш мир состоит из процедур. Время течет, что-то постоянно происходит. С утра вы можете пройти процедуру пробуждения, чистки зубов, приготовления кофе, одевания. Эти действия можно смоделировать в процедурных приложениях. Поскольку вам известен порядок событий, можно написать код для каждого действия, а затем связать их вместе — одно за другим.

Мир также имеет объектно-ориентированную природу. К объектам, с которыми вы взаимодействуете, можно отнести, например, жену и кота, старую машину и кучу велосипедных запчастей в гараже, ваше сердцебиение и график тренировок. Каждый из этих объектов обладает характерным только для него поведением. И хотя некоторые варианты взаимодействия объектов могут быть предсказуемы, может случиться так, что жена неожиданно наступит на кота, вызвав у всех учащенное сердцебиение и побудив вас пересмотреть график своих тренировок.

В мире объектов новые варианты поведения возникают вполне естественным образом. Вам не приходится создавать конкретный код для процедуры `spouse_steps_on_cat`, в ходе которой жена наступает на кота, — вам достаточно шагающего объекта-жены и объекта-кота, которому не нравится, когда на него

наступают. Поместите оба этих объекта в комнату — и станут появляться неожиданные комбинации поведения.

Книга посвящена проектированию объектно-ориентированных программ, в ней автор пытается взглянуть на мир как на серию спонтанных взаимодействий объектов. Объектно-ориентированное проектирование требует переосмысления мира: нужно отойти от его представления как коллекции предопределенных процедур и перейти к его моделированию в виде серии сообщений между объектами.

Просчеты в ООП могут казаться просчетами в самих программах, но на самом деле это просчеты в представлении окружающей обстановки. Первое требование при изучении способов ООП — погружение в суть объектов. Как только будет выработан объектно-ориентированный взгляд на окружающий мир, все остальное приложится само собой.

Эта книга проведет вас через процесс погружения. В начале первой главы описаны общие вопросы ООП. От рассмотрения доводов в пользу проектирования мы перейдем к изучению того, когда его следует проводить и оценивать. В конце главы дан краткий обзор ОПП с определением понятий, используемых в книге.

Хвала проектированию

Программное обеспечение не создается по мановению волшебной палочки. Суть в том, что приложение, будь то обычная игра или программа по управлению лучевой терапией, должно иметь определенную цель. Если бы самым экономически эффективным способом создания работоспособных приложений было программирование на пределе всех сил и возможностей разработчиков, то тем бы пришлось постоянно стоически преодолевать моральные трудности или же подыскивать себе другую работу.

К счастью, вам не придется выбирать между удовольствием, получаемым от работы, и ее продуктивностью. Технологии программирования, позволяющие писать код с удовольствием, аналогичны технологиям, позволяющим создавать приложения наиболее эффективным образом. Так что технологии ООП решают как моральные, так и технические дилеммы программирования. Четкое следование этим технологиям позволит создавать экономически

эффективные программы, к тому же над их кодом будет весьма приятно работать.

Проблемы, решаемые с помощью проектирования

Представьте, что создается новое приложение. И в отношении него уже выработан полный и выверенный набор требований. Представьте себе еще один момент: после того как приложение будет создано, в него никогда не придется вносить изменения. В этом случае проектирование не требуется. Подобно жонглеру, который крутит тарелки на гибких прутьях в мире без силы трения и гравитации, вы можете запрограммировать приложение на движение, а затем стоять в сторонке, наблюдая, как оно будет работать. Как бы ни трясло окружающий мир, «тарелки» вашего кода не снизят темпа вращения, балансируя в процессе каждого оборота, но никогда не падая на землю. Пока что-либо не изменится.

К сожалению, что-то *должно* измениться. Изменения происходят всегда. Клиенты не знают, чего хотят, нечетко выражают свои мысли. Вы порой не понимаете, что им нужно, зато знаете, как можно улучшить код. Даже самые совершенные приложения нестабильны. Приложение имело огромный успех, а теперь всем захотелось большего. Таким образом, изменения неизбежны, без них обойтись не удастся.

Изменения в требованиях являются в программировании эквивалентом трения и гравитации. Это силы, совершенно неожиданно влияющие на самые продуманные планы.

Приложения, легко поддающиеся изменениям, приятно создавать и комфортно расширять. Они обладают гибкостью и приспособляемостью. А приложения, сопротивляющиеся изменениям, совершенно иные: любое нововведение дается с большими затратами, и каждое из них приводит к подорожанию следующего изменения. Вряд ли найдется такое слабо поддающееся изменениям приложение, с которым было бы приятно работать. Худшие из них постепенно превращаются в чей-то кошмар с несчастным программистом в главной роли, который должен метаться от одной «тарелки» к другой в попытке не разбить их.

Почему изменения так нелегко даются

Объектно-ориентированные приложения составлены из взаимодействующих частей. Этими частями являются *объекты*, а взаимодействие осуществляется путем обмена *сообщениями*. Чтобы направить правильное сообщение в адрес нужного объекта, объект-отправитель сообщения должен знать, что представляет собой объект-получатель. Эти знания создают зависимости между двумя объектами, и они же мешают изменениям.

Объектно-ориентированное проектирование заключается в *управлении зависимостями*. Это набор методик создания программного кода, выстраивающих зависимости таким образом, чтобы объекты можно было изменять. При отсутствии проектирования неуправляемые зависимости порождают хаос, поскольку объекты слишком осведомлены друг о друге. Изменения, вносимые в один объект, заставляют изменять и те объекты, которые с ним сотрудничают, что, в свою очередь, заставляет вносить изменения в объекты, которые сотрудничают с этими объектами, *и так до бесконечности*. Казалось бы, незначительное усовершенствование может привести к настоящей цепной реакции, и в конечном счете код оставляют нетронутым.

Когда объекты слишком много знают, они многого требуют. Они слишком капризны, и им нужно, чтобы все было «именно так и никак иначе». Эти требования стесняют их работу. Объекты сопротивляются своему повторному использованию в различных контекстах, их очень трудно тестировать — и неизбежно возникает необходимость в их дублировании.

В небольшом приложении еще можно смириться с недостатками проектирования. Даже если все объекты взаимодействуют друг с другом, пока вы способны удерживать это в голове, у нас еще есть возможность усовершенствования приложения. Проблема, связанная со слабым проектированием *небольших* приложений, заключается в том, что в случае успеха они разрастаются до слабо спроектированных *больших* приложений. Они постепенно становятся топью, в которую вы боитесь вступить, поскольку в ней можно запросто утонуть. Простейшие правки могут вызвать каскад изменений по всему приложению, повсеместно нарушая код и требуя большого объема переделок. Тестирование попадает под перекрестный огонь и начинает казаться помехой, а не помощником.

Определение проектирования

Каждое приложение представляет собой набор кода, а систематизация этого кода называется *проектированием*. Два отдельных программиста могут решать одну и ту же задачу разными способами. Проектирование — не сборочная линия, где одинаково обученные работники создают одинаковые виджеты; это студия, где творцы-единомышленники ваяют пользовательские приложения. Проектирование — это искусство, суть которого заключается в систематизации кода.

Некоторые трудности проектирования связаны с тем, что у каждой задачи есть две составляющие. Вы должны не просто написать код для функции, которую планируете поставить сегодня, но сделать его поддающимся дальнейшим изменениям. Для любого периода времени, прошедшего с момента поставки бета-версии, стоимость изменений в конечном итоге превзойдет исходную стоимость приложения. Поскольку принципы проектирования связаны между собой и каждая проблема предполагает изменение сроков готовности проекта, проблемы проектирования могут иметь огромное количество возможных решений.

От вас требуется выработать общий взгляд на вещи (то есть вы должны сопоставить задачу, которую будет решать ваше приложение, со всеми издержками и выгодами проектировочных альтернатив), а затем разработать такую структуру кода, которая была бы экономически выгодна в настоящее время и продолжала быть таковой в будущем.

Может показаться, что прогнозирование не имеет ничего общего с программированием. Но это не так. Конечно, при проектировании не надо стараться предвидеть неизвестные требования и выбирать одно из них для применения в приложении. Программисты — не физики. В практическом проектировании не строятся прогнозы насчет того, что может случиться с вашим приложением, а просто допускается неизбежность перемен, а также их неожиданность. Дело не в выстраивании догадок, а в дополнительных возможностях, сохраняющих способность приспосабливаться к будущему. И дело не в выборе, а в свободе маневра.

Таким образом, основной задачей проектирования является снижение затрат на внесение изменений.

Инструменты проектирования

Проектирование — это не следование определенному набору правил, а путешествие по развилкам, где ранее сделанные выборы закрывают доступ к одним вариантам и открывают его к другим. При создании приложения приходится блуждать по лабиринту требований, а каждый поворот требует принятия решения, которое будет иметь последствия в будущем.

У скульптора под рукой резцы и напильники, а проектировщику объектно-ориентированных программ инструментами служат принципы и шаблоны.

Принципы проектирования

Акроним SOLID, введенный в обиход Майклом Физерсом (Michael Feathers) и популяризированный Робертом Мартином (Robert Martin), обозначает пять самых известных принципов объектно-ориентированного проектирования: единственной обязанности — **S**ingle Responsibility, открытости-закрытости — **O**pen-Closed, подстановки, предложенной Барбарой Лисков (Barabara Liskov), — **L**iskov Substitution, разделения интерфейса — **I**nterface Segregation, инверсии зависимостей — **D**ependency Inversion. К числу других принципов можно отнести DRY (Don't Repeat Yourself — «не повторяйтесь»), введенный Энди Хантом (Andy Hunt) и Дэйвом Томасом (Dave Thomas), и закон Деметры — Law of Demeter (LoD) из проекта «Деметра» Северо-Восточного университета (США).

Сами принципы будут рассматриваться по всей вкниге, а пока зададимся вопросом, откуда они взялись. Существуют ли реальные доказательства их значимости, или это просто чье-то мнение, которым можно пренебречь? И вообще, с какой стати их следует придерживаться?

Все эти принципы возникли в результате выборов, сделанных кем-то при написании кода. На ранних стадиях развития объектно-ориентированной технологии программисты подметили, что одни структуры кода упрощали им жизнь, а другие — усложняли. Исходя из этого опыта было выработано мнение о том, как создается качественный код.

Со временем подключились преподаватели с научными степенями, которые решили дать количественную оценку «совершенству». Такое рвение похвально. Если что-то можно подсчитать, то есть вычислить *количественные показатели* относительно нашего кода, и соотнести эти показатели с приложениями высокого или низкого качества (для которых также нужны объективные показатели), то нам удастся чаще создавать продукты с меньшей себестоимостью. Возмож-

ность оценки качества изменит ООП, переведя его из разряда бесконечно оспариваемых мнений в разряд поддающейся измерению науки.

Именно это и было сделано в 1990-х годах Чидамбером (Chidamber), Кемерером (Kemerer)[1] и Базили (Basili)[2]. Они взяли объектно-ориентированные приложения и попытались дать количественную оценку коду. Они придумали названия и систему измерений, описав общий размер классов, их вложенность друг в друга, глубину и ширину иерархии наследования, а также количество методов, вовлекаемых в результате отправки любого сообщения. Они подобрали структуры кода, которые, по их мнению, могли иметь значение, вывели формулы подсчета, а затем сопоставили получившиеся показатели с качеством имеющихся приложений. Их исследование показывает определенную взаимосвязь между этими технологиями и высококачественным кодом.

Хотя кажется, что эти исследования подтверждают принципы проектирования, любой опытный программист воспринимает их с оговоркой. В ходе этих исследований, проведенных на ранней стадии, были изучены весьма небольшие по объему приложения, созданные аспирантами, и лишь этого одного уже достаточно, чтобы настороженно относиться к сделанным выводам. Код в этих приложениях мог не давать общей картины, свойственной объектно-ориентированным приложениям реального мира.

Но оказалось, что опасения напрасны. В 2001 году Лэйнг (Laing) и Колеман (Coleman) изучили ряд приложений, созданных в NASA Goddard Space Flight Center, в попытке отыскать в них «способ производства более дешевых и высококачественных программных продуктов»[3]. Они изучили три приложения разного качества, в одном из которых было 1617 классов и более 500 000 строк кода. Их исследования подкрепили результаты более ранних исследований и еще раз подтвердили важность принципов проектирования.

Даже если вы никогда не слышали об этих исследованиях, можете быть уверены в достоверности их выводов. Принципы разумного проектирования — это факты, поддающиеся конкретным оценкам. Следуя им, вы повысите качество своего кода.

---

1 *Chidamber S. R., Kemerer C. F.* A metrics suite for object-oriented design // IEEE Trans. Softw. Eng, 1994. — P. 476–493.

2 Basili Technical Report // A Validation of Object-Oriented Design Metrics as Quality Indicators. — Univ. of Maryland, Dep. of Computer Science, College Park, M. D., 20742 USA. — April 1995.

3 *Laing V., Coleman C.* Principal Components of Orthogonal Object-Oriented Metrics. — 2001.