Содержание

ГЛАВА 8

Принцип открытости/закрытости

Завершив изучение этой главы, вы сможете выполнять следующие действия:

- понимать разные интерпретации принципа открытости/закрытости;
- трактовать код SOLID как допускающий только добавление;
- сравнивать друг с другом разные механизмы определения точек расширения в классах;
- использовать в качестве руководящего указания для точек расширения устойчивость к изменениям.

Двоякий характер принципа открытости/закрытости вызывает путаницу. Его отточенное название наводит на мысль о коде, который является одновременно и разрешающим, и запрещающим. А тот факт, что существует несколько вариаций его определения, еще больше усугубляет положение.

Было бы беспечно отдать предпочтение одному определению перед другим и применять исключительно его. Вместо этого в главе имеющиеся определения и их последствия сравниваются в попытке добраться до самой сути данного принципа. Цель заключается в том, чтобы предложить максимально полезные рекомендации, которые позволят создавать код, более адаптивный к будущим изменениям.

Введение в принцип открытости/закрытости

Есть два определения принципа открытости/закрытости, которые должны быть изучены: первоначальное из 1980-х годов и более современное. Второе определение стремится уточнить первое, придавая ему больше контекста и проясняя границы действия принципа.

Определение Мейера

Бертран Мейер в своей книге *Object-Oriented Software Construction* (Prentice Hall, 1988 год) описал принцип открытости/закрытости (open/closed principle — OCP) следующим образом.

> *Программные сущности должны быть открыты для расширения, но закрыты для модификации.*

Определение Мейера чаще всего и цитируют, но существует второе определение, которое предложил Мартин.

Определение Мартина

Роберт С. Мартин определял принцип OCP в многочисленных работах, написанных за несколько лет. В противоположность краткому оригиналу он выбрал более многословное определение.

> *Модули, которые согласованы с принципом открытости/закрытости, обладают двумя основными характеристиками.*
>
> 1. *Они открыты для расширения.*
>
>   *Это означает, что поведение модуля может быть расширено. Когда требования приложения изменяются, мы способны расширить модуль новыми линиями поведения, которые удовлетворяют возникшим изменениям. Другими словами, у нас есть возможность изменить то, что делает модуль.*
>
> 2. *Они закрыты для модификации.*
>
>   *Расширение поведения модуля не ведет к изменениям в исходном или двоичном коде модуля. Двоичная исполняемая версия модуля, будь она библиотекой DLL или JAR-файлом, остается незатронутой.*
>
> Роберт С. Мартин,
> *Гибкая разработка программ на Java и C++.*
> *Принципы, паттерны и методики*
> ("Диалектика", 2016 год)

Для обеих сторон принципа открытости/закрытости Мартин подробно раскрывает смысл основных терминов в определении Мейера. Как поясняет Мартин, чтобы код был открытым для расширения, разработчики обязаны быть способными реагировать на изменение требований и поддерживать новые функциональные средства. Это должно достигаться, невзирая на закрытость модулей для модификации. Разработчики обязаны поддерживать новую функциональность, не редактируя исходный код — или скомпилированную сборку — существующих модулей.

Прежде чем приступить к выяснению, как такое возможно, следует отметить, что есть исключения из ограничивающего пункта "они закрыты для модификации" принципа OCP, которые временами упоминаются: изменения с целью исправления ошибок или дефектов и изменения, которые могут делаться без осведомленности со стороны клиентов.

Исправления дефектов

Дефекты — всеобщая проблема в ПО, которую полностью предотвратить невозможно. Встретив их, вы должны отреагировать исправлением проблемного кода. Несомненно, исправление влечет за собой модификацию существующего класса — если только вы не захотите создать дубликат класса и реализовать исправление дефекта в новой версии. Но это звучит излишне запутанно и противоречит руководящему принципу оставаться на стороне прагматизма, а не чистоты.

Ниже обрисован двухшаговый процесс устранения дефекта.

1. Напишите отказывающий модульный и/или интеграционный тест, который специально нацелен на дефект. Для этого потребуются надежные и повторяемые шаги по воссозданию условий, при которых код отказывает. В случае унаследованного кода, т.е. кода без тестов, обратитесь в главу 6 за сведениями о том, как реализовать тест золотого мастера, чтобы получить в свое распоряжение набор тестов. Кроме того, вспомните синтаксис AAA модульного теста, представленный в главе 5. Вы должны быть способны *организовать* тестируемую систему, чтобы перевести ее в состояние, где может проявиться дефект, выполнить специфическое *действие*, в котором скрыт дефект, и в заключение выдвинуть *утверждение* относительно *ожидаемого* поведения. После написания такой тест первоначально не пройдет. Это демонстрирует тот факт, что все дефекты являются следствием отсутствующих тестов. Если есть тест, который обнаруживает дефект, тогда он не проходит.
2. *Модифицируйте* исходный код, чтобы модульный тест прошел. Исключение из принципа OCP, касающееся исправления дефектов, становится на данном этапе жизненно необходимым из-за того, что без него не было бы возможности модифицировать любой существующий код. Редактирование тестируемой системы позволяет перевести отказывающий тест из красного в зеленое состояние — от отказа к успеху. Когда вы удостоверитесь, что остальные тесты не перестали проходить как побочный эффект, дефект считается устраненным.

Осведомленность клиентов

Более либеральное исключение из правила “они закрыты для модификации” связано с тем, что в существующий код допускается вносить любые изменения до тех пор, пока они не требуют модификации любого клиента данного кода. Это акцентирует внимание на том, каким образом связаны программные модули на всех уровнях детализации: классы с классами, сборки со сборками и подсистемы с подсистемами.

Если изменение в одном классе вызывает изменение в другом, тогда говорят, что два класса *тесно* связаны. И наоборот, если класс можно изменять изолированно, не принуждая к изменениям других классов, то участвующие классы являются *слабо* связанными. На всех уровнях всегда предпочтительнее иметь

слабое связывание. Сохранение слабого связывания ограничивает влияние, которое оказывает принцип OCP, если разрешена модификация существующего кода, не вызывающая внесения дополнительных изменений в клиенты.

Точки расширения

Теперь, когда правило "они закрыты для модификации" принципа OCP прояснилось, можно обсудить правило "они открыты для расширения". Классы, соблюдающие принцип OCP, должны быть открытыми для расширения за счет того, что в них определены точки расширения, в которых будущая функциональность может привязываться к существующему коду и предоставлять новые линии поведения.

В данном разделе рассматриваются возможные варианты для разных видов точек расширения, а также исследуются все аргументы за и против. Мы продолжим пользоваться примером класса `TradeProcessor` из предшествующих глав, на этот раз с точки зрения клиента, взаимодействующего с классом.

Код без точек расширения

Прежде всего, на что похож код, не имеющий точек расширения? На рис. 8.1 показано, что происходит, когда класс без точек расширения нуждается в новой функциональности.

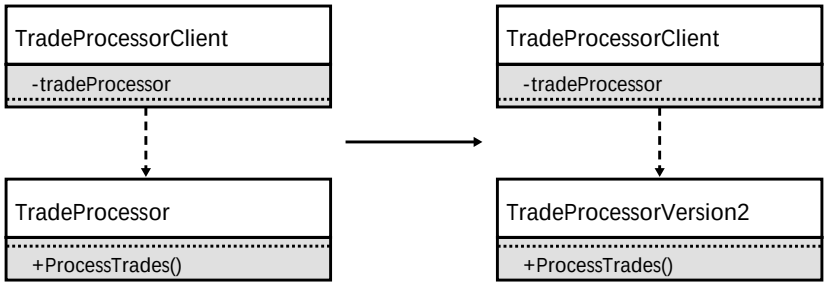

Рис. 8.1. Когда точек расширения нет, клиенты придется изменять

Класс `TradeProcessorClient` зависит непосредственно от класса `TradeProcessor`. Поступает новое требование, в результате которого возникает необходимость внести изменения в класс `TradeProcessor`. Не имея возможности модифицировать исходный класс, придется создать новую версию (`TradeProcessorVersion2`), которая содержит новую функциональность, реализующую новое требование. Поскольку клиент напрямую зависит от класса `TradeProcessor` и по причине отсутствия в `TradeProcessor` точек расширения новую функциональность понадобится поместить внутрь нового класса. Побочным эффектом такого изменения становится потребность в редактировании класса `TradeProcessorClient`, чтобы он зависел от нового класса. Похоже, проще было бы разрешить редактирование существующего класса `TradeProcessor`, не так ли?

Если в существующий код разрешено вносить изменения, которые не оказывают никакого влияния на клиентов, тогда возможно не придется создавать полностью новую версию `TradeProcessor`. Если бы изменилась сигнатура метода `ProcessTrades()`, то это было бы не просто изменением реализации класса, но также и изменением его интерфейса. Все изменения интерфейса вызывают изменения в клиентах, т.к. клиенты тесно связаны с интерфейсами своих зависимостей.

Виртуальные методы

Альтернативная реализация класса `TradeProcessor` содержит точку расширения: метод `ProcessTrades()` является *виртуальным*. На рис. 8.2 видно, как теперь организованы три класса.

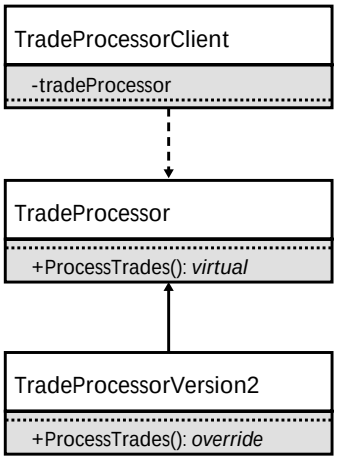

Рис. 8.2. Клиент зависит от класса `TradeProcessor`, который может быть расширен через наследование

Любой класс, в котором один из членов помечен как виртуальный, открыт для расширения. Расширение такого типа достигается через *наследование реализации*. Когда поступает требование ввести в метод `ProcessTrades()` класса `TradeProcessor` новую функциональность, можно создать подкласс существующего класса `TradeProcessor`, не модифицируя его исходный код, и изменить в подклассе метод `ProcessTrades()`.

В этом случае класс `TradeProcessorClient` в изменении не нуждается, потому что с помощью полиморфизма клиента можно снабдить новой версией `TradeProcessor` и вынудить его вызывать реализацию метода `ProcessTrades()` из подкласса.

Однако масштаб повторной реализации кое в чем ограничен. Доступ к базовому классу имеется, так что вызывать `TradeProcessor.ProcessTrades()` можно, но изменять отдельные строки исходного метода нельзя. Исходный метод либо вызывается целиком, возможно с добавлением новой функциональности до или после вызова, либо реализуется полностью заново. Никаких компромиссов с виртуальным методом не существует. Не забывайте, что подклассы могут обращаться только к тем членам базового класса, которые помечены как защищенные. Если бы класс `TradeProcessor` был определен с многочисленными закрытыми членами, тогда доступ к ним отсутствовал бы, а внесение изменений в исходный класс, вполне естественно запрещено принципом OCP.

Абстрактные методы

Более гибкой точкой расширения, которая задействует наследование реализации, является *абстрактный* метод. В данном случае `TradeProcessor` — абстрактный класс, определяющий открытый метод `ProcessTrades()`, который делегирует выполнение алгоритма обработки трем защищенным абстрактным

методам. Клиент совершенно не осведомлен об этих защищенных методах, а ввиду того, что они абстрактные, никакой реализации не предоставляется. На рис. 8.3 показаны отношения между участвующими классами.

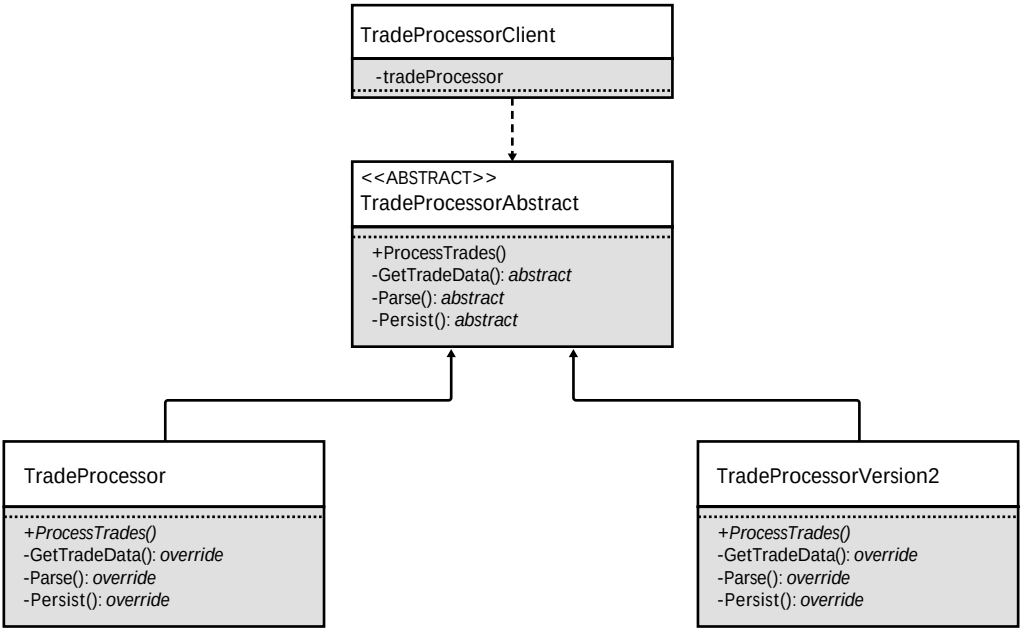

Рис. 8.3. Абстрактные методы предоставляют точки расширения для будущих подклассов

Определены две версии обработчика сделок. Обе они наследуют метод `ProcessTrades()` прямо из абстрактного базового класса и обе предлагают собственные реализации для абстрактных методов. Клиент зависит от абстрактного класса, так что можно предоставить любой из имеющихся двух конкретных подклассов — или новый подкласс для новых требований — и принцип OCP не будет нарушен.

Это пример *паттерна "Шаблонный метод"* (Template Method), когда моделируется алгоритм, общие шаги которого допускают настройку благодаря делегированию абстрактным методам. Фактически базовый класс делегирует выполнение отдельных шагов процесса подклассам.

Такие методы не обязаны быть абстрактными; они могли бы быть виртуальными. Разница связана главным образом со *степенью детализации* того, что было сделано расширяемым. Вместо замены метода `ProcessTrades()` как единого целого теперь можно заменять избранные части процесса, переопределяя составляющие его методы.

Наследование интерфейса

Последний тип точки расширения, обсуждаемый в главе, представляет собой альтернативу наследованию реализации: *наследование интерфейса*. Здесь

зависимость клиента от *класса* заменяется уже знакомым делегированием работы интерфейсу. На рис. 8.4 демонстрируется зависимость клиента от интерфейса и две реализации данного интерфейса.

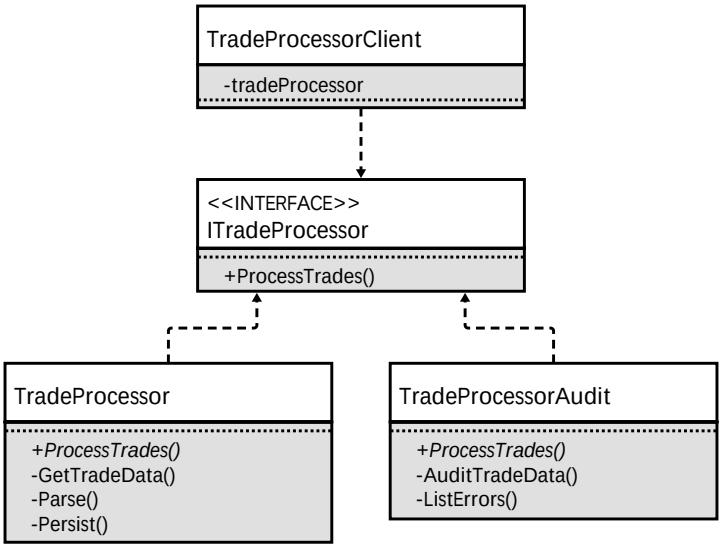

Рис. 8.4. Клиент зависит от интерфейса, а не от класса

Наследование интерфейса предпочтительнее наследования реализации. При наследовании реализации все подклассы — настоящие и будущие — являются *клиентами*, что препятствует модификации, поскольку из-за наследования реализации подклассы зависят также от *реализации*. Таким образом, клиенты потенциально осведомлены обо всех изменениях в реализации. Это перекликается с рекомендацией отдавать предпочтение композиции перед наследованием и сохранять иерархии наследования неглубокими, с небольшим количеством уровней подклассов. Если изменение с целью добавления какого-то члена делается в верхней части диаграммы наследования, тогда оно повлияет на всех участников иерархии.

Интерфейсы также считаются лучшими точками расширения, потому что их можно декорировать обильными графами объектов с функциональностью, которая касается множества разных контекстов. Вдобавок они гибче классов. Я вовсе не имею в виду, что виртуальные и абстрактные методы, которые формируют точки расширения при наследовании классов, не являются полезными, просто они и близко не обеспечивают того уровня адаптации, какого можно добиться посредством интерфейсов.

"Проектируйте наследование или запретите его"

Вот как высказался о наследовании Джошуа Блох в своей книге *Effective Java* (Addison-Wesley, 2008 год).

Проектируйте и документируйте наследование или запретите его.

Если вы решили использовать в качестве точки расширения наследование реализации, тогда должны надлежащим образом проектировать и документировать класс, чтобы подстраховать и проинформировать программистов, которые в будущем займутся расширением класса. Наследование классов может быть сложным — новые подклассы способны нарушить работу существующего кода непредсказуемыми путями.

Очень важно отметить, что любой класс, который не помечен ключевым словом `sealed`, заявляет о своей поддержке наследования. Чтобы побуждать к созданию своих подклассов, класс не обязательно должен быть абстрактным или содержать виртуальные методы. С помощью ключевого слова `new` унаследованные члены можно скрывать, но это препятствует полиморфизму, открыто пренебрегая ожиданиями.

Запрет наследования за счет запечатывания класса дает четкий сигнал другим программистам, которые могут использовать класс: его наследование не просто не поддерживается, оно не допускается. В итоге исчезнет соблазн попытаться расширить класс, и программисты направят свои силы на поиски альтернативы.

Устойчивость к изменениям

Теперь вы обеспечены несколькими инструментами, помогающими соблюдать принцип OCP. Вам известно, в каких обстоятельствах можно редактировать существующие классы, и вы знаете, что для поддержки будущих изменений в требованиях понадобится реализовать точки расширения в своем коде. Вы также осведомлены о возможности применения интерфейсов в качестве точек расширения, чтобы сделать свой код по-настоящему адаптируемым.

Недостающим ингредиентом является знание того, *когда* и *где* использовать данный принцип. Доводя ситуацию до логической крайности, должны ли вы добавлять точки расширения повсюду и все время? Сделает ли это код бесконечно гибким или же здесь действует своеобразный эквивалент закона убывающей доходности?

Именно тут жизненно важным становится еще один принцип, связанный с OCP: *устойчивость к изменениям*. Термин придумал Алистер Кокберн.

> *Идентифицируйте места предсказуемых изменений и создайте вокруг них стабильный интерфейс.*
>
> Алистер Кокберн,
> *Pattern Languages of Program Design, vol. 2*
> (Addison-Wesley, 1996 год)

Несколько сбивает с толку то, что определение ссылается на *предсказуемые изменения*, но сам принцип называется *устойчивостью к изменениям*. Тем не менее, “предсказуемые изменения”, по моему мнению, более точный термин. Два главных аспекта приведенного определения заслуживают более пристального внимания.

Предсказуемые изменения

Требования к отдельному классу должны быть напрямую связаны с требованиями бизнес-клиента. Если эта связь игнорируется, тогда возникает риск, что класс не будет служить ни одной из целей, которые запрашивал бизнес-клиент. В ходе спринта пользовательские истории берутся из беклога спринта, и разработчики ведут беседу с владельцем продукта. На данной стадии должны задаваться вопросы относительно возможного будущего, касающегося требований. В результате наполняются смыслом *предсказуемые изменения*, которые могут быть переведены в точки расширения.

Стабильный интерфейс

Даже если вы делегируете работу только интерфейсам, клиенты по-прежнему зависят от этих интерфейсов. Если интерфейсы изменяются, тогда клиенты также должны изменяться. Основное преимущество зависимости от интерфейсов заключается в том, что они будут изменяться с гораздо меньшей вероятностью, чем реализации. Если интерфейс помещен в сборку, отдельную от его реализации, то клиенты могут изменяться без влияния на реализации интерфейса, а реализации могут изменяться, не влияя на клиентов.

Безусловно, очень важно, чтобы все интерфейсы, выбранные для представления точки расширения, были стабильными. Вероятность и частота изменений интерфейсов должны быть низкими, иначе для использования новой версии придется вносить изменения во все клиенты.

Только достаточная степень адаптации

Существует понятие “зоны обитаемости” (или “зоны Златовласки”), где код содержит только правильное количество точек расширения и в правильных местах, чтобы сделать возможным внесение изменений в областях, в которых они нужны, не увеличивая сложность и не допуская чрезмерной работы по проектированию решения. Для любой отдельно взятой задачи может быть слишком малая, чересчур большая и *только достаточная* степень адаптации.

Программисты, начинающие свою карьеру, склонны писать код, который стоит ближе к процедурному, даже на объектно-ориентированных языках, таких как C#. Они тяготеют к применению классов как хранилищ для методов, не обращая внимания на то, действительно ли эти методы сочетаются друг с другом. В коде нет различимой архитектуры, а точек расширения немного (да и те могут оказаться в неподходящих местах). Любые изменения в требованиях неизбежно влекут за собой прямые изменения в существующем классе или классах. Так был организован первоначальный класс `TradeProcessor` в главе 7. Он представлял собой “божественный объект ”, которому прекрасно известно все, что нужно делать в программе.

Однако временами такое решение считается корректным. Если вы оцените предсказуемые изменения для такого мелкого инструмента, как `TradeProcessor`, и придете к заключению, что он вряд ли изменится каким-

либо образом, тогда первоначальной версии кода будет достаточно. А может быть достаточно окажется версии, которая была подвергнута рефакторингу для повышения ясности. Дополнительное время, уделенное рефакторингу для абстрагирования, будет потрачено понапрасну, если вы никогда не задействуете предоставленные точки расширения. Мало того, намного труднее рассуждать о коде и его работе, когда он разнесен по разным файлам и сборкам с реализациями, скрытыми позади интерфейсов.

На противоположном конце спектра находится программист, который начал абстрагирование кода с использованием интерфейсов. Он словно открыл для себя молоток и теперь для него все выглядит похожим на гвоздь. Производимый им код представляет собой густую смесь точек расширения, большинство из которых никогда не понадобится. Требуется прикладывать значительные усилия, чтобы собрать вместе код, который постоянно делегирует обязанности интерфейсам, и в первую очередь значительные усилия были нужны для написания такого кода.

Если бы указанные два архетипа программистов — и их код — удалось объединить, тогда результатом был бы гармоничный компромисс, когда имеется достаточное количество точек расширения и код способен к адаптации в областях, где требования неясны, изменчивы или трудны для реализации. Тем не менее, это приходит с опытом. Достичь такого дзен-подобного состояния предсказуемых изменений непросто без того, чтобы начать как наивный новичок и постепенно эволюционировать в превосходно абстрагирующего всезнайку!

Предсказуемые изменения или гипотетическое обобщение

Разработка ПО представляет собой развивающуюся инженерную практику, до сих пор не ставшую на ноги как наука. По этой причине можно свободно интерпретировать, что именно является жестким правилом, а что — советом, рекомендацией или передовым опытом.

К двум рекомендациям такого рода относятся *предсказуемые изменения* и *гипотетическое обобщение*.

Рекомендация, касающаяся предсказуемых изменений, устанавливает необходимость явного указания того, что разрешено и что запрещено расширять. Рекомендация относительно гипотетического обобщения утверждает, что вы должны остерегаться попыток заранее определить, как класс может применяться для решения общей задачи, чтобы не создать утечку абстракции.

Обе рекомендации кажутся расходящимися друг с другом, но могут ли они быть двумя сторонами одной и той же монеты?

Подумайте, сколько раз вы использовали ключевое слово `sealed` для класса или делали класс зависимым непосредственно от другого класса, а не интерфейса. Теперь сравните это с количеством раз, когда вы либо создавали зависимость от интерфейса, либо помечали класс как `abstract`, а метод как `abstract` или `virtual`.

Когда класс запечатан, в будущем создавать его подклассы невозможно, что предотвращает применение класса в качестве точки расширения. Абстрактные классы предназначены для расширения и не допускают создания экземпляров без первоначального построения своих подклассов. То же самое справедливо для абстрактных методов. Виртуальные методы предоставляют содержательную стандартную реализацию, но впоследствии могут быть каким-то образом специализированы.

Все это варианты, выбираемые при проектировании. Одни из них являются неявными и непреднамеренными: вы просто не помечаете класс как `sealed`; следовательно, вы можете обойти определенные методы в будущем. Другие являются явными. В идеале все проектные варианты должны выбираться с явным намерением.

Нужно ли так много интерфейсов?

В настоящее время интерфейсы — вездесущая конструкция в объектно-ориентированном программировании. Интерфейсы не задействуют для предоставления точек расширения лишь в немногих случаях. Но интерфейсы представляют собой предельно ясную точку расширения: клиенты получают способ передачи сообщения какому-то другому объекту и появляются неограниченные возможности обработки такого сообщения.

В некоторых случаях их ценность очевидна. Обычно это касается тех мест в приложении, где требуется развязывание клиента и службы. Распространенным и реальным примером является доступ к данным, когда клиентскому коду необходимо обращаться к данным, которые хранятся и извлекаются по идентификатору. Такой класс снабжен интерфейсом, представляющим способ доступа к данным. Впоследствии этот интерфейс реализуется специфическим механизмом хранения данных: реляционной базой данных, документно-ориентированным хранилищем, кешем типа “ключ-значение”, файловым хранилищем на диске, словарем в памяти, имитированным тестовым дублером, обращением к службе... Список можно продолжить.

Однако есть случаи, когда интерфейсы и неисчислимые возможности реализации неоправданны. Возьмем, к примеру, отображение кода с одного уровня на другой. Предположим, что код содержит уровень доступа к данным, который извлекает объекты передачи данных, специфичные для реализации. На уровне модели предметной области нужно получать данные, хранящиеся в объектах передачи данных, в распознаваемом формате. Это делает необходимым код отображения, но будет ли много реализаций такого кода или только одна каноническая версия, для которой явно запрещено расширение?

Ожидаемыми возвращаемыми значениями из хранилища являются объекты предметной области. Таким образом, преобразование специфичных к реализации объектов передачи данных в типы модели предметной области — забота конкретной реализации уровня доступа к данным. Следовательно, уровни отображения концептуально располагаются вместе с их реализациями доступа к данным, а не где-то между уровнем доступа к данным и уровнем модели предметной области.

Если специфическая реализация уровня доступа к данным зависит от конкретного типа объекта передачи данных и служит определенной модели предметной области, тогда от предоставления расширяемого и заменяемого кода отображения не будет никакой пользы, а одни лишь издержки: пониженная читабельность.

В рассмотренном примере было показано, как избежать гипотетического обобщения, прогнозируя *отсутствие* изменений. Две рекомендации не являются взаимоисключающими: они дополняют друг друга. Если искомое обобщение оказывается гипотетическим, тогда прогнозируется отсутствие специфических изменений. Если же искомое обобщение оказывается не гипотетическим, то должны быть спрогнозированы специфические изменения.

Заключение

Принцип открытости/закрытости — это генеральная линия для всего проектирования классов и интерфейсов, а также способа построения разработчиками кода, который разрешает изменения с течением времени. С каждым пройденным спринтом неизбежно появляются новые требования, которые должны приниматься во внимание. Тем не менее, признание того, что в изменениях нет ничего плохого — лишь полдела. Если написанный до сих пор код не строился с расчетом на изменения, тогда внесение изменений будет трудным, долгим и подверженным ошибкам.

Гарантируя, что код открыт для расширения, но закрыт для модификации, вы фактически запрещаете будущие изменения в существующих классах и сборках, что заставляет программистов создавать новые классы, которые могут подключаться к точкам расширения. Доступны два главных типа точек расширения: наследование реализации и наследование интерфейса. Виртуальные и абстрактные методы позволяют создавать подклассы, которые настраивают методы из базового класса. Если классы делегируют обязанности интерфейсам, то это обеспечивает более высокую гибкость в организации точек расширения посредством разнообразных паттернов.

Однако недостаточно лишь знать о возможности встраивания точек расширения в код. Необходимо также понимать, когда это применимо. Концепция устойчивости к изменениям предполагает, что вы идентифицируете части требований, которые вероятно будут изменяться или особенно трудны для реализации, и поместите их за точками расширения. Код может быть определен довольно жестко, с небольшими границами для расширения или совершенствования, или же он может быть очень гибким, с несметным количеством точек расширения, готовых к поддержке новых требований. Любой из указанных вариантов может быть корректным в зависимости от имеющегося сценария и контекста.