5 Непрерывное улучшение API

> Нет необходимости меняться. Выживание необязательно.
>
> *У. Эдвардс Деминг*

В предыдущей главе мы ознакомились с жизненным циклом API и определили десять столпов работы, на которых следует сосредоточиться. Жизненный цикл и эти столпы определяют, что именно необходимо сделать для выпуска вашего первого API. Кроме того, столпы важны для всех изменений, которые вы будете вносить в период жизненного цикла уже опубликованного API. Управление изменениями в API — это важнейшая часть стратегии управления им в целом.

Изменения в API могут серьезно повлиять на ваше программное обеспечение, саму программу и опыт пользователей. Внесение в код изменений, которые повредят API, может даже создать катастрофическую цепную реакцию, влияющую на все компоненты, использующие этот код. Изменения, которые не вредят API, также способны вызвать большие затруднения, если изменяют интерфейс неожиданным образом. Продукты с API создают сложные группы взаимозависимостей. Поэтому управление изменениями — важный раздел в управлении API.

Если бы в опубликованные API не надо было вносить изменения, управлять ими было бы довольно легко. Но, разумеется, это неотъемлемая часть активно применяемых API. В какой-то момент нужно будет исправить ошибку, улучшить опыт разработчика или оптимизировать код реализации. Эти задачи требуют вмешательства в используемый API.

Управление изменениями API усложняется большим объемом работы. Продукт с API — это не только интерфейс. Он состоит из многих частей: интерфейсов, кода, данных, документации, инструментов и процессов. Все эти части продукта API можно изменять и управлять ими надо осторожно.

Управлять изменениями API непросто, но это необходимо. К тому же возможность внесения изменений дает больше свободы. Если бы опубликованный API

нельзя было изменять, было бы гораздо сложнее выпускать изначальный релиз. Приходилось бы применять к разработке API правила конструирования и запуска космических ракет. Понадобилось бы потратить очень много времени и сил на предварительное планирование и сделать серьезные вложения в разработку, чтобы убедиться, что этот API сможет проработать долго. Пришлось бы сначала учесть все возможные неполадки и предотвратить их.

К счастью, вам не обязательно работать по такой схеме. Конечно, если вы добавите в свой API возможность вносить изменения, это может принести большую выгоду. Более дешевые и простые изменения означают, что их можно делать чаще. Это позволяет вам больше рисковать, потому что появляется возможность быстрее исправлять неполадки. Таким образом, вы можете чаще улучшать API.

В этой главе мы представим философию непрерывного улучшения для API с возможностью внесения изменений. Вы узнаете, как непрерывная серия небольших поэтапных изменений может стать наилучшим способом улучшения вашего продукта с API, а также почему API так сложно изменять и как можно увеличить их способность к изменениям. Но прежде, чем углубиться в эти темы, необходимо лучше разобраться с тем, что означает выражение «изменения в API».

Изменения в API

В главе 1 мы обозначили разницу между составляющими продукта с API: интерфейсом, реализацией и готовым экземпляром. После публикации API необходимо будет управлять изменениями всех этих составляющих. Иногда придется изменять все вместе, но может оказаться, что вы будете изменять какие-то из этих элементов API независимо от других. В данном разделе мы рассмотрим влияние изменений, происходящих в каждой из этих частей. И даже добавим новый тип элементов API под названием «ресурсы поддержки», которые включают в себя части API, используемые только для расширения опыта разработчика.

Эти четыре типа изменений в API будут взаимно влиять друг на друга. Они формируют комплекс взаимозависимых изменений: изменения в модели интерфейса будут иметь далеко идущие последствия, в то же время ресурсы поддержки можно легко изменять отдельно от остальных элементов. Когда мы начнем исследовать каждый из типов изменений, вы станете лучше понимать причины существования этих взаимозависимостей.

Жизненный цикл релиза API

Программное обеспечение становится другим, когда в него вносятся изменения. Действия, предпринимаемые для того, чтобы сделать правильные изменения в кратчайшее время и с наилучшим качеством, — это процесс релиза. Как и у программного обеспечения, у API тоже есть процесс релиза — ряд действий для внесе-

ния изменений. Мы называем этот процесс жизненным циклом из-за циклического характера изменений: пока одно осуществляется, следующее уже готово. Понимать, как протекает жизненный цикл релиза, очень важно, потому что он сильно влияет на возможность изменения вашего API.

Каждое изменение, которое вы вносите в API, должно быть реализовано. Жизненный цикл релиза — это набор действий, которые приводят к его реализации. Он определяет, как изменение из идеи превращается в осуществленную и поддерживаемую часть системы. Жизненный цикл релиза объединяет все столпы, описанные нами в главе 4, в последовательность согласованной работы.

Если жизненный цикл релиза медленный, количество изменений API уменьшится. Если в нем не гарантируется качество, изменять API будет более рискованно. Если он отклоняется от требований к изменениям, последние будут менее полезными. Важно понимать, как протекает жизненный цикл релиза. Плюс в том, что у API он не отличается от жизненного цикла программного обеспечения или поставки компонентов системы. Таким образом, вы можете применять существующие руководства для релизов программного обеспечения к компонентам, из которых состоит API. Рассмотрим самые популярные из них.

Один из самых известных жизненных циклов программного обеспечения — это традиционный *жизненный цикл разработки систем* (system development lifecycle, SDLC). В той или иной форме он существует с 1960-х годов. Он определяет набор этапов для разработки и выпуска системы программного обеспечения. Число и название используемых этапов могут меняться, но обычно этот набор выглядит следующим образом: подготовка, анализ, разработка, конструирование, тестирование, реализация и техническая поддержка.

Если идти по ступеням SDLC по порядку, это будет разработка программного обеспечения по *каскадной* модели. На самом деле Уинстон Ройс изобрел не каскадную модель, но теперь данный тип жизненного цикла называют именно так. Это означает, что каждая фаза SDLC должна быть закончена до начала следующего этапа. Таким образом, изменение проходит с верхней ступени последовательно через каждую следующую ступень.

Один из недостатков каскадной модели состоит в том, что нужен высокий уровень уверенности в требованиях и вообще в этой области, потому что непросто иметь дело с большим количеством изменений в технических заданиях. Если у вас с этим проблемы, можно использовать циклический процесс разработки программного обеспечения. *Циклический* SDLC позволяет команде разработки производить несколько циклов релизов для одного множества требований. Каждый цикл выполняет подмножество требований, чтобы в результате последовательности всех циклов все требования были выполнены.

Если развивать идею циклов, мы придем к *спиральному* SDLC. В этом типе цикла программное обеспечение разрабатывается, конструируется и тестируется в цикличных стадиях, и каждый цикл потенциально может выполнить изначальные

требования. В спиральном SDLC воплощается дух гибкой методологии разработки и метода SCRUM.

Мы рассмотрели три распространенные формы жизненного цикла программного обеспечения. У каждой из них есть свои плюсы и минусы, и вам нужно будет выбрать подходящий жизненный цикл релиза. В этой книге мы пытаемся дать вам возможность использовать любой стиль. Говоря об изменениях, будем упоминать ваш жизненный цикл релиза, но не станем навязывать порядок, в котором должны производиться базовые действия, или применяемый жизненный цикл программного обеспечения. Вместо этого сосредоточимся на улучшении продукта, доступном благодаря жизненному циклу релиза. Но прежде, чем перейти к этой теме, подробнее поговорим о типах изменений в API, которые придется поддерживать жизненному циклу вашего релиза.

Изменение модели интерфейса

У каждого API есть модель интерфейса. Это информация, описывающая поведение API с точки зрения *пользователя*. Она рассматривает набор абстрактных понятий, определяющих, как API будет работать, и включает в себя подробности протоколов связи, сообщений и терминологии. Отличительной чертой модели API является то, что она не воплощена в жизнь, — это абстракция и ее невозможно использовать в компьютерной системе.

Хотя модель интерфейса нельзя задействовать с помощью программного обеспечения, ею можно поделиться с пользователями. Для этого модель должна быть сохранена или *выражена*. Например, можно выразить модель интерфейса, нарисовав квадратики и линии на маркерной доске. Такую нарисованную модель нельзя *задействовать*, но как абстракция она поможет вашей команде работать над дизайном API.

Конечно, модели интерфейса — это не только рисунки на доске и наброски на салфетках. Их также можно выражать с помощью языков на основе моделей или даже с помощью кода приложения. Например, Open-API Specification — популярный стандартизированный язык для описания моделей интерфейса. Использование стандартизированного языка для моделирования дает вам бонус — систему инструментов, которая поможет уменьшить затраты на реализацию вашей модели.

Рисовать или составлять модель можно как угодно — не существует никаких правил, регламентирующих уровень детализации или формат передачи модели. Но имейте в виду, что метод, который вы выберете для выражения модели, сильно повлияет на уровень детализации и описания. Маркерные доски и техника свободного рисования предоставляют максимальную свободу мысли, но ограничены физическими размерами и возможностью реализовать модели. Языки описания API обеспечивают более быстрый путь к реализации, но ограничивают свободу жестким синтаксисом и терминологией.

Столп «Дизайн» жизненного цикла нашего API фокусируется на создании и изменении модели интерфейса, поэтому большая часть описанной нами работы идеально ему подходит. Но она связана не только с дизайном. На самом деле большая часть столпов жизненного цикла зависят от модели интерфейса. Это происходит потому, что они тоже являются выражениями этой модели.

Допустим, вы выразили модель интерфейса в виде рисунка на доске или с помощью языка Open-API — и так же будете выражать ее с помощью кода приложения, документации по API и модели данных. Когда интерфейс уже опубликован и разработчики начинают писать использующий его код, они своей реализацией тоже выражают вашу модель. Все эти выражения модели подразумевают отношения взаимозависимости. Вот почему изменения в модели так важны.

Предметно-ориентированное проектирование

Идея программного обеспечения, ориентированного на модель, где реализация является выражением этой модели, пришла к нам из подхода к разработке ПО, созданного Эриком Эвансом, — предметно-ориентированного проектирования (domain-driven design, DDD). Если вы еще не читали его книгу Domain-Driven Design: Tackling Complexity in the Heart of Software[1] (Addison-Wesley), стоит сделать это!

Лучшие продукты с API имеют единообразные модели интерфейса. Поэтому разработчикам не приходится разрешать конфликты между документацией и уже опубликованными API из-за того, что выражаемые ими модели как-то различаются. Это стремление к единообразию усложняет внесение изменений в модели интерфейса, потому что их необходимо синхронизировать во всем продукте.

Использование единообразной модели не означает, что код реализации и внутренняя база данных должны задействовать ту же модель, что и интерфейс вашего API. На самом деле обычно применять одну и ту же модель для интерфейса, кода и данных — неудачная идея: то, что идеально подходит пользователям интерфейса, не обязательно идеально работает внутри вашей собственной реализации. Единообразная модель означает, что внутренние части вашей реализации API нужно будет перевести в эту модель интерфейса, прежде чем они «достигнут поверхности» API.

Изменения в модели интерфейса имеют огромное влияние на систему, но они неизбежны для любого активно используемого продукта с API. Может понадобиться добавить техподдержку новой функции, изменить что-то, чтобы упростить применение API, или, возможно, убрать устаревшую часть интерфейса, потому что ваша бизнес-модель кардинально изменилась. Из-за всех взаимозависимостей

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011.

изменения модели интерфейса всегда потенциально могут повлиять на код в приложениях, задействующих этот API.

Потенциальное влияние изменений модели интерфейса на пользователей API во многом связано с уровнем связанности между их кодом и вашим интерфейсом. Если особенностью API, которые вы создаете и реализуете, является слабая связанность, то даже при крупных изменениях эффект окажется меньшим. Например, использование API с событийно-ориентированным стилем или стилем гипермедиа имеет преимущество — более слабую связанность между клиентским кодом и API. В случае событийно-ориентированной системы вы можете менять алгоритм соответствия шаблону, не внося при этом никаких изменений в компонент, отправляющий события. API с гипермедиа может позволить вам совершать манипуляции с обязательными свойствами для инициирования работы, не меняя клиентский код, совершающий запрос.

Выбор подходящего стиля интерфейса может помочь уменьшить затраты на изменения его модели. Но такая увеличившаяся изменяемость API тоже не бесплатна. Чтобы они работали, придется создать подходящую инфраструктуру и реализации на обеих сторонах — клиентской и серверной. Зачастую ограничения и окружение, в котором вы работаете, сужают ваш выбор: например, разработчики, которые пишут клиентское ПО для вашего API, могут не иметь опыта написания приложений в стиле гипермедиа. В таких случаях придется просто смириться с тем, что изменения модели интерфейса будут дорого стоить.

Лучший способ уменьшить внешнее влияние изменений модели — вносить их *до того*, как интерфейс будет опубликован. Джошуа Блох, разработчик Java Collections API, сказал: «Общедоступные API, как бриллианты, — вечны» (https://www.infoq.com/articles/API-Design-Joshua-Bloch). Как только вы открываете интерфейс для пользователей, внесение изменений значительно усложняется. Мудрый владелец продукта с API по максимуму загружает изменения в модель интерфейса на стадии дизайна, чтобы не платить за них после публикации API.

Изменения в реализации

Реализация API — это модель интерфейса, выраженная с помощью компонентов, которые воплотят ее в жизнь. Она позволяет другим компонентам программного обеспечения использовать этот интерфейс. Реализация API включает в себя код, конфигурацию, данные, инфраструктуру и даже выбор протокола. Эти компоненты обычно являются *скрытыми* частями продукта — они заставляют API работать, но мы не обязаны делиться их деталями с пользователями.

Невозможно опубликовать API без реализации, и ее придется изменять в течение всего времени существования API. Поскольку реализация воплощает модель интерфейса, ее потребуется менять при каждом изменении модели. Но иногда у вас

будет возможность изменить реализацию независимо от модели. Например, может понадобиться исправить ошибку в коде реализации, уменьшить время ожидания медленного API или даже полностью переписать код, потому что он просто вам больше не нравится.

В тех случаях, когда изменения в реализации не привязаны к модели, их эффект скрыт за интерфейсом API. Таким образом, пользователям не придется ничего менять, чтобы воспользоваться этими обновлениями. Это не означает, что на них это никак не *повлияет*, — например, оптимизация работы приложения может сильно повлиять на восприятие этой работы конечным пользователем. Но так вы можете избежать управления изменениями в клиентском ПО, которое зависит от вашего API. В общем, изменения в реализации могут быть сделаны и после публикации API, не требуя изменений в модели интерфейса.

Риск внесения независимых изменений в реализацию состоит в том, что они могут ухудшить надежность, единообразие и доступность продукта. Например, если изменения в коде портят работу экземпляра API или реализация начинает отличаться от своей документации, ваши клиентские приложения пострадают. Изменения в реализации могут повлиять на готовый экземпляр и ресурсы поддержки API, поэтому каждый из этих элементов надо согласованно обновлять, тестировать и одобрять.

Изменение экземпляра

Как мы уже рассказали, реализация API выражает модель в виде интерфейса, который можно использовать. Но это возможно, только когда она запущена на устройстве в сети и доступна для пользовательских приложений. *Готовый экземпляр* API — это управляемое работающее выражение модели интерфейса, доступное для применения вашей целевой аудиторией.

Любые изменения в модели интерфейса или реализации потребуют соответствующих изменений в готовых экземплярах. Изменения в API нельзя считать внесенными, пока не обновлены экземпляры, используемые пользовательскими приложениями. Однако экземпляр API можно менять независимо, не затрагивая при этом модель или реализацию. Это может быть простой случай изменения конфигурационного значения или что-то более сложное, например дублирование и удаление работающего экземпляра. Такие изменения влияют только на свойства системы во время действия программы, прежде всего на доступность, наблюдаемость, надежность и качество работы.

Чтобы уменьшить влияние на систему независимых изменений в готовом экземпляре, необходимо заранее продумать дизайн системной архитектуры. Мы обсудим свойства системы и самые важные факторы позже, когда начнем говорить о системе API.

Изменения в ресурсах поддержки

Если API — это продукт, он должен быть не просто запущенным на сервере кодом, выражающим модель. Из главы 1 мы узнали, что поддержка разработчиков, которые должны использовать наши API, — это важная часть философии «API как продукт». Создание позитивного опыта разработчика почти всегда требует каких-либо ресурсов поддержки за пределами реализации интерфейса. Например, в эти ресурсы могут входить документация по API, регистрация разработчиков, инструменты для выявления неполадок и раздачи важных материалов и даже сотрудники техподдержки, которые будут помогать разработчикам решать проблемы.

В течение всего времени работы API вам нужно будет обновлять и улучшать материалы, процессы и сотрудников, которые поддерживают продукт. Зачастую это окажется результатом каскадных изменений модели интерфейса, реализации или экземпляра API. Ресурсы поддержки, находящиеся ниже по течению, будут затронуты при изменениях любой части API. Таким образом, затраты на изменения API вырастут с ростом количества ресурсов поддержки опыта разработчика.

Независимые изменения можно вносить и в ресурсы поддержки, например изменить внешний вид страницы с документацией в процессе модернизации. Такие изменения сильно влияют на опыт разработчика вашего продукта с API, но не на модель интерфейса, реализацию или готовый экземпляр — точнее, могут влиять, но не напрямую, а в результате того, что программой начинают чаще пользоваться и больше интересоваться.

Изменения в ресурсах поддержки вызывают наименьший каскадный эффект, но также могут привести к наивысшим затратам, потому что больше всех зависят от других элементов API. Снижение затрат на эти изменения может принести большие дивиденды с точки зрения общих затрат на изменения в продукте с API. Поэтому имеет смысл вкладываться в дизайн, инструменты и автоматизацию, чтобы тратить меньше сил на изменение этих ресурсов.

Непрерывное управление изменениями

Мы описали четыре типа изменений, с которыми вы столкнетесь в работе с API: изменения модели интерфейса, реализации, экземпляра и ресурсов. Все они поразному влияют на API и пользовательские приложения, поэтому необходимо внимательно управлять изменениями, чтобы точно не снизить качество своего продукта.

Если применять подход AaaP, эти изменения можно считать попытками улучшить продукт с API, а не просто изменениями ради изменений. Таким образом, все время, потраченное на изменения в интерфейсе, реализации, экземпляре или ресурсах, можно оправдать улучшением опыта разработчика или снижением затрат спонсоров на поддержку продукта.