

Сергей Константинов. API

Source:



[Лицензия Creative Commons](https://creativecommons.org/licenses/by-nc/4.0/)

Это произведение доступно по [лицензии Creative Commons «Attribution-NonCommercial» \(«Атрибуция — Некоммерческое использование»\) 4.0 Всемирная](https://creativecommons.org/licenses/by-nc/4.0/).

О структуре этой книги

Книга, которую вы держите в руках, состоит из трех больших разделов.

В первом разделе мы поговорим о проектировании API на стадии разработки концепции - как грамотно выстроить архитектуру, от крупноблочного планирования до конечных интерфейсов.

Второй раздел будет посвящён жизненному циклу API - как интерфейсы эволюционируют со временем и как развивать продукт так, чтобы отвечать потребностям пользователей.

Наконец, третий раздел будет касаться больше не-разработческих сторон жизни API - поддержки, маркетинга, работы с комьюнити.

Первые два будут интересны скорее разработчикам, третий — и разработчикам, и менеджерам. При этом мы настаиваем, что как раз третий раздел — самый важный для разработчика API. Ввиду того, что API - продукт для разработчиков, перекладывать ответственность за его развитие и поддержку на не-разработчиков неправильно: никто кроме вас самих не понимает так хорошо продуктовые свойства вашего API.

На этом переходим к делу.

Введение. Определение API

Прежде чем говорить о разработке API, необходимо для начала договориться о том, что же такое API. Энциклопедия скажет нам, что API — это программный интерфейс приложений. Это точное определение, но бессмысленное. Примерно как определение человека по Платону: «двуногое без перьев» — определение точное, но никоим образом не дающее нам представление о том, чем на самом деле человек примечателен. (Да и не очень-то и точное: Диоген Синопский как-то ощипал петуха и заявил, что это человек Платона; пришлось дополнить определение уточнением «с плоскими ногтями».)

Что же такое API по смыслу, а не по формальному определению?

Вероятно, вы сейчас читаете эту книгу посредством браузера. Чтобы браузер смог отобразить эту страничку, должны корректно отработать: разбор URL согласно спецификации; служба DNS; соединение по протоколу TLS; передача данных по протоколу HTTP; разбор HTML-документа; разбор CSS-документа; корректный рендеринг HTML+CSS.

Но это только верхушка айсберга. Для работы HTTP необходима корректная работа всего сетевого стека, который состоит из 4-5, а то и больше, протоколов разных уровней. Разбор HTML-документа производится согласно сотням различных спецификаций. Рендеринг документа обращается к нижележащему API операционной системы, а также напрямую к API видеокарты. И так далее, и тому подобное — вплоть до того, что наборы команд современных CISC-процессоров имплементируются поверх API микрокоманд.

Иными словами, десятки, если не сотни, различных API должны отработать корректно для выполнения базовых действий типа просмотра web-страницы; без надёжной работы каждого из них современные информационные технологии попросту не могли бы существовать.

API — это обязательство. Формальное обязательство связывать между собой различные программируемые контексты.

Когда меня просят привести пример хорошего API, я обычно показываю фотографию римского виадука:

- он связывает между собой две области
- обратная совместимость нарушена ноль раз за последние две тысячи лет.

Отличие римского виадука от хорошего API состоит лишь в том, что API предлагает *программный* контракт. Для связывания двух областей необходимо написать некоторый

код. Цель этой книги — помочь вам разработать API, так же хорошо выполняющий свою задачу, как и римский виадук.

Виадук также хорошо иллюстрирует другую проблему разработки API: вашими пользователями являются инженеры. Вы не поставляете воду напрямую потребителю: к вашей инженерной мысли подключаются заказчики путём пристройки к ней каких-то своих инженерных конструкций. С одной стороны, вы можете обеспечить водой гораздо больше людей, нежели если бы вы сами подводили трубы к каждому крану. С другой — качество инженерных решений заказчика вы не можете контролировать, и проблемы с водой, вызванные некомпетентностью подрядчика, неизбежно будут валить на вас.

Поэтому проектирование API налагает на вас несколько большую ответственность. **API является как мультипликатором ваших возможностей, так и мультипликатором ваших ошибок.**

Критерии качества API

Прежде чем излагать рекомендации, нам следует определиться с тем, что мы считаем «хорошим» API, и какую пользу мы получаем от того, что наше API «хорошее».

Начнём со второго вопроса. Очевидно, «хорошесть» API определяется в первую очередь тем, насколько он помогает разработчикам решать стоящие перед ними задачи. (Можно резонно возразить, что решение задач, стоящих перед разработчиками, не обязательно влечёт за собой выполнение целей, которые мы ставим перед собой, предлагая разработчикам API. Однако манипуляция общественным мнением не входит в область интересов автора этой книги: здесь и далее предполагается, что API существует в первую очередь для того, чтобы разработчики решали с его помощью свои задачи, а не ради каких-то не декларируемых явно целей.)

Как же дизайн API может помочь разработчику? Очень просто: API должно *решать задачи*, и делать это максимально удобно и понятно. Путь разработчика от формулирования своей задачи до написания работающего кода должен быть максимально коротким. Это, в том числе, означает, что:

- из структуры вашего API должно быть максимально очевидно, как решить ту или иную задачу; в идеале разработчику должно быть достаточно одного взгляда на документацию, чтобы понять, с помощью каких сущностей следует решать его задачу;
- API должно быть читаемым: в идеале разработчик, просто глядя в номенклатуру методов, сразу пишет правильный код, не углубляясь в детали (особенно — детали реализации!); немаловажно уточнить, что из интерфейсов объектов должно быть понятно не только решение задачи, но и возможные ошибки и исключения;

- API должно быть консистентно: при разработке новой функциональности, т.е. при обращении к каким-то незнакомым сущностям в API, разработчик может действовать по аналогии с уже известными ему концепциями API, и его код будет работать.

Однако статическое удобство и понятность API — это простая часть. В конце концов, никто не стремится специально сделать API нелогичным и нечитаемым — всегда при разработке мы начинаем с каких-то понятных базовых концепций. При минимальном опыте проектирования сложно сделать ядро API, не удовлетворяющее критериям очевидности, читаемости и консистентности.

Проблемы начинаются, когда мы начинаем API развивать. Добавление новой функциональности рано или поздно приводит к тому, что некогда простое и понятное API становится наслоением разных концепций, а попытки сохранить обратную совместимость приводят к нелогичным, неочевидным и попросту плохим решениям. Отчасти это связано так же и с тем, что невозможно обладать полным знанием о будущем: ваше понимание о «правильном» API тоже будет меняться со временем, как в объективной части (какие задачи решает API и как лучше это сделать), так и в субъективной — что такое очевидность, читабельность и консистентность для вашего API.

Принципы, которые я буду излагать ниже, во многом ориентированы именно на то, чтобы API правильно развивалось во времени и не превращалось в нагромождение разнородных неконсистентных интерфейсов. Важно понимать, что такой подход тоже бесплатен: необходимость держать в голове варианты развития событий и закладывать возможность изменений в API означает избыточность интерфейсов и возможно излишнее абстрагирование. И то, и другое, помимо прочего, усложняет и работу программиста, пользующегося вашим API. **Закладывание перспектив «на будущее» имеет смысл, только если это будущее у API есть, иначе это попросту оверинжиниринг.**

Обратная совместимость

Обратная совместимость — это некоторая *временная* характеристика качества вашего API. Именно необходимость поддержания обратной совместимости отличает разработку API от разработки программного обеспечения вообще.

Разумеется, обратная совместимость не абсолютна. В некоторых предметных областях выпуск новых обратно несовместимых версий API является вполне рутинной процедурой. Тем не менее, каждый раз, когда выпускается новая обратно несовместимая версия API, всем разработчикам приходится инвестировать какое-то ненулевое количество усилий, чтобы адаптировать свой код к новой версии. В этом плане выпуск новых версий API является некоторого рода «налогом» на потребителей

— им нужно тратить вполне осязаемые деньги только для того, чтобы их продукт продолжал работать.

Конечно, крупные компании с прочным положением на рынке могут позволить себе такой налог взимать. Более того, они могут вводить какие-то санкции за отказ от перехода на новые версии API, вплоть до отключения приложений.

С нашей точки зрения, подобное поведение ничем не может быть оправдано. Избегайте скрытых налогов на своих пользователей. Если вы можете не ломать обратную совместимость — не ломайте её.

Да, безусловно, поддержка старых версий API — это тоже своего рода налог. Технологии меняются, и, как бы хорошо ни было спроектировано ваше API, всего предусмотреть невозможно. В какой-то момент ценой поддержки старых версий становится невозможность предоставлять новую функциональность и поддерживать новые платформы, и выпустить новую версию всё равно придётся. Однако вы по крайней мере сможете убедить своих потребителей в необходимости перехода.

Более подробно о политиках версионирования будет рассказано в разделе II.

О версионировании

Здесь и далее мы будем придерживаться принципов версионирования [semver](#):

1. Версия API задаётся тремя цифрами, вида 1.2.3
2. Первая цифра (мажорная версия) увеличивается при обратно несовместимых изменениях в API
3. Вторая цифра (минорная версия) увеличивается при добавлении новой функциональности с сохранением обратной совместимости
4. Третья цифра (патч) увеличивается при выпуске новых версий, содержащих только исправление ошибок

Выражения «мажорная версия API» и «версия API, содержащая обратно несовместимые изменения функциональности» тем самым следует считать эквивалентными.

Замечание о терминологии

Разработка программного обеспечения характеризуется, помимо прочего, существованием множества различных парадигм разработки, адепты которых зачастую настроены весьма воинственно по отношению к адептам других парадигм. Поэтому при написании этой книги мы намеренно избегаем слов «метод», «объект», «функция» и так далее, используя нейтральный термин «сущность». Под «сущностью» понимается

некоторая атомарная единица функциональности — класс, метод, объект, монада, прототип (нужное подчеркнуть).

I. Проектирование API

1. Пирамида контекстов API

Подход, который мы используем для проектирования, состоит из четырёх шагов:

- определение области применения;
- разделение уровней абстракции;
- разграничение областей ответственности;
- описание конечных интерфейсов.

Этот алгоритм строит API сверху вниз, от общих требований и сценариев использования до конкретной номенклатуры сущностей; фактически, двигаясь этим путем, вы получите на выходе готовое API — чем этот подход и ценен.

Может показаться, что наиболее полезные советы и best practice приведены в последнем разделе, однако это не так; цена ошибки, допущенной на разных уровнях весьма различна. Если исправить плохое именование довольно просто, то исправить неверное понимание того, зачем вообще нужно API, практически невозможно.

NB. Здесь и далее мы будем рассматривать концепции разработки API на примере некоторого гипотетического API заказа кофе в городских кофейнях. На всякий случай сразу уточним, что пример является синтетическим; в реальной ситуации, если бы такое API пришлось проектировать, оно вероятно было бы совсем не похоже на наш выдуманный пример.

2. Определение области применения

Ключевой вопрос, который вы должны задать себе четыре раза, выглядит так: какую проблему мы решаем? Задать его следует четыре раза с ударением на каждом из четырёх слов.

1. *Какую* проблему мы решаем? Можем ли мы чётко описать, в какой ситуации гипотетическим потребителям-разработчикам нужно наше API?
2. Какую *проблему* мы решаем? А мы правда уверены, что описанная выше ситуация — проблема? Действительно ли кто-то готов платить (в прямом и переносном смысле) за то, что ситуация будет как-то автоматизирована?
3. Какую проблему *мы* решаем? Действительно ли решение этой проблемы находится в нашей компетенции? Действительно ли мы находимся в той позиции, чтобы решить эту проблему?

4. Какую проблему мы *решаем*? Правда ли, что решение, которое мы предлагаем, действительно решает проблему? Не создаём ли мы на её месте другую проблему, более сложную?

Итак, предположим, что мы хотим предоставить API автоматического заказа кофе в городских кофейнях. Попробуем применить к ней этот принцип.

1. Зачем кому-то может потребоваться API для приготовления кофе? В чем неудобство заказа кофе через интерфейс, человек-человек или человек-машина? Зачем нужна возможность заказа машина-машина?
- Возможно, мы хотим решить проблему выбора и знания? Чтобы человек наиболее полно знал о доступных ему здесь и сейчас опциях.
 - Возможно, мы оптимизируем время ожидания? Чтобы человеку не пришлось ждать, пока его заказ готовится.
 - Возможно, мы хотим минимизировать ошибки? Чтобы человек получил именно то, что хотел заказать, не потеряв информацию при разговорном общении либо при настройке незнакомого интерфейса кофе-машины.

Вопрос «зачем» — самый важный из тех вопросов, которые вы должны задавать себе. Не только глобально в отношении целей всего проекта, но и локально в отношении каждого кусочка функциональности. **Если вы не можете коротко и понятно ответить на вопрос «зачем эта сущность нужна» — значит, она не нужна.**

Здесь и далее предположим (в целях придания нашему примеру глубины и некоторой упорности), что мы оптимизируем все три фактора в порядке убывания важности.

2. Правда ли решаемая проблема существует? Действительно ли мы наблюдаем неравномерную загрузку кофейных автоматов по утрам? Правда ли люди страдают от того, что не могут найти поблизости нужный им латте с ореховым сиропом? Действительно ли людям важны те минуты, которые они теряют, стоя в очередях?
3. Действительно ли мы обладаем достаточным ресурсом, чтобы решить эту проблему? Есть ли у нас доступ к достаточному количеству кофемашин и клиентов, чтобы обеспечить работоспособность системы?
4. Наконец, правда ли мы решим проблему? Как мы поймём, что оптимизировали перечисленные факторы?

На все эти вопросы, в общем случае, простого ответа нет. В идеале ответы на эти вопросы должны даваться с цифрами в руках. Сколько конкретно времени тратится неоптимально, и какого значения мы рассчитываем добиться, располагая какой плотностью кофемашин? Заметим также, что в реальной жизни просчитать такого рода

цифры можно в основном для проектов, которые пытаются влезть на уже устоявшийся рынок; если вы пытаетесь сделать что-то новое, то, вероятно, вам придётся ориентироваться в основном на свою интуицию.

Почему API?

Т.к. наша книга посвящена не просто разработке программного обеспечения, а разработке API, то на все эти вопросы мы должны взглянуть под другим ракурсом: а почему для решения этих задач требуется именно API, а не просто программное обеспечение? В нашем вымышленном примере мы должны спросить себя: зачем нам нужно предоставлять сервис для других разработчиков, чтобы они могли готовить кофе своим клиентам, а не сделать своё приложение для конечного потребителя?

Иными словами, должна иметься веская причина, по которой два домена разработки ПО должны быть разделены: есть оператор(ы), предоставляющий API; есть оператор(ы), предоставляющий сервисы пользователям. Их интересы в чем-то различны настолько, что объединение этих двух ролей в одном лице нежелательно. Более подробно мы изложим причины и мотивации делать именно API в разделе II.

Заметим также следующее: вы должны браться делать API тогда и только тогда, когда в ответе на второй вопрос написали «потому что в этом состоит наша экспертиза». Разрабатывая API вы занимаетесь некоторой мета-разработкой: вы разрабатываете ПО для того, чтобы другие могли разрабатывать ПО для решения задачи пользователя. Не обладая экспертизой в обоих этих доменах (API и конечные продукты) написать хорошее API сложно.

Для нашего умозрительного примера предположим, что в недалеком будущем произошло разделение рынка кофе на две группы игроков: одни предоставляют само железо, кофейные аппараты, а другие имеют доступ к потребителю — примерно как это произошло, например, с рынком авиабилетов, где есть собственно авиакомпании, осуществляющие перевозку, и сервисы планирования путешествий, где люди выбирают варианты перелётов. Мы хотим агрегировать доступ к железу, чтобы владельцы приложений могли встраивать заказ кофе.

Что и как

Закончив со всеми теоретическими упражнениями, мы должны перейти непосредственно к дизайну и разработке API, имея понимание по двум пунктам:

1. Что конкретно мы делаем
2. Как мы это делаем

В случае нашего кофепримера мы:

1. Предоставляем сервисам с большой пользовательской аудиторией API для того, чтобы их потребители могли максимально удобно для себя заказать кофе.
2. Для этого мы абстрагируем за нашим HTTP API доступ к «железу» и предоставим методы для выбора вида напитка и места его приготовления и для непосредственно исполнения заказа.

С этими вводными мы можем переходить непосредственно к разработке.

Разделение уровней абстракции

«Разделите свой код на уровни абстракции» - пожалуй, самый общий совет для разработчиков программного обеспечения. Однако будет вовсе не преувеличением сказать, что изоляция уровней абстракции — самая сложная задача, стоящая перед разработчиком API.

Прежде чем переходить к теории, следует чётко сформулировать, *зачем* нужны уровни абстракции и каких целей мы хотим достичь их выделением.

Вспомним, что программный продукт - это средство связи контекстов, средство преобразования терминов и операций одной предметной области в другую. Чем дальше друг от друга эти области отстоят - тем большее число промежуточных передаточных звеньев нам придётся ввести. Вернёмся к нашему примеру с кофейнями. Какие уровни сущностей мы видим?

1. Непосредственно состояние кофе-машины и шаги приготовления кофе. Температура, давление, объём воды.
2. У кофе есть мета-характеристики: сорт, вкус, вид напитка.
3. Мы готовим с помощью нашего API *заказ* — один или несколько стаканов кофе с определенной стоимостью.
4. Наши кофе-машины как-то распределены в пространстве (и времени).
5. Кофе-машина принадлежит какой-то сети кофейен, каждая из которых обладает какой-то айдентикой и специальными возможностями.

Каждый из этих уровней задаёт некоторый срез нашего API, с которым будет работать потребитель. Выделяя иерархию абстракций мы прежде всего стремимся снизить связность различных сущностей нашего API. Это позволит нам добиться нескольких целей:

1. Упрощение работы разработчика и легкость обучения: в каждый момент времени разработчику достаточно будет оперировать только теми сущностями, которые нужны для решения его задачи; и наоборот, плохо выстроенная изоляция приводит к тому, что разработчику нужно держать в голове множество концепций, не имеющих прямого отношения к решаемой задаче.

2. Возможность поддерживать обратную совместимость; правильно подобранные уровни абстракции позволят нам в дальнейшем добавлять новую функциональность, не меняя интерфейс.
3. Поддержание интероперабельности. Правильно выделенные низкоуровневые абстракции позволят нам адаптировать наше API к другим платформам, не меняя высокоуровневый интерфейс.

Допустим, мы имеем следующий интерфейс:

- GET /recipes/lungo
— возвращает рецепт лунго;
- POST /coffee-machines/orders?machine_id={id}
{recipe:"lungo"}
— размещает на указанной кофе-машине заказ на приготовление лунго и возвращает идентификатор заказа;
- GET /orders?order_id={id}
— возвращает состояние заказа;

И зададимся вопросом, каким образом разработчик определит, что заказ клиента готов. Допустим, мы сделаем так: добавим в рецепт лунго эталонный объём, а в состояние заказа — количество уже налитого кофе. Тогда разработчику нужно будет проверить совпадение этих двух цифр, чтобы убедиться, что кофе готов.

Такое решение выглядит интуитивно плохим, и это действительно так: оно нарушает все вышеперечисленные принципы:

1. Для решения задачи «заказать лунго» разработчику нужно обратиться к сущности «рецепт» и выяснить, что у каждого рецепта есть объём. Далее, нужно принять концепцию, что приготовление кофе заканчивается в тот момент, когда объём сравнялся с эталонным. Нет никакого способа об этой конвенции догадаться: она неочевидна и её нужно найти в документации. При этом никакой пользы для разработчика в этом знании нет.
2. Мы автоматически получаем проблемы, если захотим варьировать размер кофе. Допустим, в какой-то момент мы захотим представить пользователю выбор, сколько конкретно миллилитров лунго он желает. Тогда нам придётся проделать один из следующих трюков:
 - или мы фиксируем список допустимых объёмов и заводим фиктивные рецепты типа /recipes/small-lungo, recipes/large-lungo. Почему фиктивные? Потому что рецепт один и тот же, меняется только объём. Нам придётся либо тиражировать одинаковые рецепты, отличающиеся только объёмом, либо

вводить какое-то «наследование» рецептов, чтобы можно было указать базовый рецепт и только переопределить объём;

- или мы модифицируем интерфейс, объявляя объём кофе, указанный в рецепте, значением по умолчанию; при размещении заказа мы разрешаем указать объём, отличный от эталонного:

```
POST /coffee-machines/orders?machine_id={id}
```

```
{recipe:"lungo","volume":"800ml"}
```

Для таких кофе произвольного объёма нужно будет получать требуемый объём не из GET /recipes, а из GET /orders. Сделав так, мы сразу получаем клубок из связанных проблем:

- разработчик, которому придётся поддержать эту функциональность, имеет высокие шансы сделать ошибку: добавив поддержку произвольного объёма кофе в код, работающий с POST /coffee-machines/orders нужно не забыть переписать код проверки готовности заказа;
- мы получим классическую ситуацию, когда одно и то же поле (объём кофе) значит разные вещи в разных интерфейсах. В GET /recipes поле «объём» теперь значит «объём, который будет запрошен, если не передать его явно в POST /coffee-machines/orders; переименовать его в «объём по умолчанию» уже не получится, с этой проблемой теперь придётся жить.

3. Вся эта схема полностью неработоспособна, если разные модели кофе-машин производят лунго разного объёма. Для решения задачи «объём лунго зависит от вида машины» нам придётся сделать совсем неприятную вещь: сделать рецепт зависимым от id машины. Тем самым мы начнём активно смешивать уровни абстракции: одной частью нашего API (рецептов) станет невозможно пользоваться без другой части (информации о кофе-машинах). Что немаловажно, от разработчиков потребуется изменить логику своего приложения: если раньше они могли предлагать сначала выбрать объём, а потом кофе-машину, то теперь им придётся полностью изменить этот шаг.

Хорошо, допустим, мы поняли, как сделать плохо. Но как же тогда сделать *хорошо*? Разделение уровней абстракции должно происходить вдоль трёх направлений:

1. От сценариев использования к их внутренней реализации: высокоуровневые сущности и номенклатура их методов должны напрямую отражать сценарии использования API; низкоуровневый - отражать декомпозицию сценариев на составные части.

Здесь мы должны явно обратиться к выписанному нами ранее «что» и «как». В идеальном мире высший уровень абстракции вашего API должен быть просто переводом записанной человекочитаемой фразы на машинный язык. Если нужно узнать, готов ли заказ — значит, должен быть метод is-order-ready (если мы считаем эту операцию действительно важной и частотной) или хотя бы GET /orders/{id}/status для того, чтобы явно узнать статус заказа. Эту логику

требуется проработать вниз до самых мелких и частных сценариев типа определения температуры напитка или наличия у исполнителя картонного держателя нужного размера.

2. От терминов предметной области пользователя к терминам предметной области исходных данных — в нашем случае от высокоуровневых понятий «рецепт», «заказ», «бренд», «кофейня» к низкоуровневым «температура напитка» и «координаты кофе-машины»
3. Наконец, от структур данных, в которых удобно оперировать пользователю к структурам данных, максимально приближенных к «сырым» - в нашем случае от «лунго» и «сети кофеен "Ромашка"» - к сырым байтовым данным, описывающим состояние кофе-машины марки «Доброе утро» в процессе приготовления напитка.

Чем дальше находятся друг от друга программные контексты, которые соединяет наше API - тем более глубокая иерархия сущностей должна получиться у нас в итоге.

В нашем примере с определением готовности кофе мы явно пришли к тому, что нам требуется промежуточный уровень абстракции:

- с одной стороны, «заказ» не должен содержать информацию о датчиках и сенсорах кофе-машины;
- с другой стороны, кофе-машина не должна хранить информацию о свойствах заказа (да и вероятно её API такой возможности и не предоставляет).

Введём промежуточный уровень: нам нужно звено, которое одновременно знает о заказе, рецепте и кофе-машине. Назовём его «уровнем исполнения»: его ответственностью является интерпретация заказа, превращение его в набор команд кофе-машине. Самый простой вариант — ввести абстрактную сущность «задание» task:

- заказ порождает одно или несколько заданий, указывая для задания конкретный рецепт и кофе-машину;
- задание в свою очередь оперирует командами кофе-машины и отвечает за интерпретацию состояния датчиков.

Таким образом, наше API будет выглядеть примерно так:

- POST /orders — создаёт заказ;
- GET /tasks?order_id={order_id} — позволяет получить список заданий по заказу.

Внимательный читатель может здесь поинтересоваться, а в чём, собственно разница по сравнению с наивным подходом? Напомню, мы рассмотрели выше примерно такой вариант:

- POST /coffee-machines/orders?machine_id={id}
 {recipe:"lungo","volume":"800ml"}
 — создаёт заказ указанного объёма
- GET /orders/{id}
 {..."volume_requested":"800ml","volume_prepared":"120ml"...}
 — состояние исполнения заказа (налито 120 мл из запрошенных 800).

По сути пара volume_requested / volume_prepared и является аналогом дополнительной сущности task, зачем мы тогда усложняли?

Во-первых, в схеме с дополнительным уровнем абстракции мы скрываем конструирование самого объекта task. Если от GET /orders/{id} ожидается, что он вернёт хотя бы логически те же параметры заказа, что были переданы в POST /coffee-machines/orders, то при конструировании task сформировать нужный набор параметров — уже наша ответственность, спрятанная внутри обработчика создания заказа. Мы можем переформулировать параметры заказа в более удобные для исполнения на кофе-машине термины — например, возвращаясь к вопросу проверки готовности, явно сформулировать политику определения готовности кофе:

- POST /tasks/?order_id={order_id}
 {..."volume_requested":"800ml","readiness_policy":"check_volume"...}
 — внутри обработчика создания заказа мы обратились к спецификации кофе-машины и поставили задачу в соответствии с ней. (Здесь мы предполагаем, что POST /tasks — внутренний метод создания задач; он может и не существовать в виде API.)
- GET /tasks/{id}/status
 {..."volume_prepared":"200ml","ready":false}
 — в публичном интерфейсе

На это (совершенно верное!) замечанием мы ответим, что выделение уровней абстракции — прежде всего *логическая* процедура: как мы объясняем себе и разработчику, из чего состоит наш API. Мы могли бы просто ограничиться выделением секции task в ответе GET /orders/{id} — или вовсе сказать, что task — это просто четверка полей (ready, volume_requested, volume_prepared, readiness_policy) и есть. **Абстрагируемая дистанция между сущностями существует объективно**, каким бы образом мы ни написали конкретные интерфейсы. Наша задача состоит только лишь в том, чтобы эта дистанция была разделена на уровни *явно*. Чем неявнее разведены уровни абстракции (или хуже того, перемешаны) уровни абстракции, тем сложнее будет разобраться в вашем API и тем хуже будет написан использующий его код.

NB: важно заметить, что с дальнейшей проработкой уровень исполнения, скорее всего, сам должен будет разделиться на два и более уровня, т.к. «задача» по сути — просто сущность-зонтик, связывающая в рамках заказа несколько высокоуровневых сущностей. Идея определения параметров кофе-машины на этапе создания заказов не

очень удобна, да и до манипуляции командами кофе-машины и состоянием сенсоров всё ещё далеко с точки зрения абстрагирования. Но мы пока оставим в таком виде, для удобства дальнейшего изложения.

Изоляция уровней абстракции

Важное свойство правильно подобранных уровней абстракции, и отсюда требование к их проектированию — это требование изоляции: **взаимодействие возможно только между сущностями соседних уровней абстракции**. Если при проектировании выясняется, что для выполнения того или иного действия требуется «перепрыгнуть» уровень абстракции, это явный признак того, что в проекте допущены ошибки.

Возвращаясь к нашему примеру с готовностью кофе: проблемы с определением готовности кофе исходя из объёма возникают именно потому, что мы не можем ожидать от пользователя, создающего заказ, знания о необходимости проверки объёма налитого реальной кофе-машиной объёма кофе. Мы вводим дополнительный уровень абстракции именно для того, чтобы на нём переформулировать, что такое «заказ готов».

Важным следствием этого принципа является то, что информацию о готовности заказа нам придётся «прорастить» через все уровни абстракции:

1. На физическом уровне мы будем оперировать состоянием кофе-машины, её сенсоров;
2. На уровне исполнения статус готовности означает, что состояние сенсоров приведено к эталонному (в случае политики "check_volume" — что налит именно тот объём кофе, который был запрошен);
3. На пользовательском уровне статус готовности заказа означает, что все ассоциированные задачи выполнены.

На каждом уровне абстракции понятие «готовность» переформулируется в терминах нижележащей предметной области, и так вплоть до физического уровня.

Аналогично нам придётся поступить и с действиями, доступными на том или ином уровне. Если, допустим, в нашем API появится метод отмены заказа `cancel`, то его придётся точно так же «спустить» по всем уровням абстракции.

- `POST /orders/{id}/cancel` работает с высокоуровневыми данными о заказе:
 - проверяет авторизацию, т.е. имеет ли право этот пользователь отменять этот заказ;
 - решает денежные вопросы — нужно ли делать рефанд
 - находит все незавершённые задачи и отменяет их
- `POST /tasks/{id}/cancel` работает с исполнением заказа:
 - определяет, возможно ли физически отменить исполнение, есть ли такая функция у кофе-машины;

- генерирует последовательность действий отмены (возможно, не только непосредственно для самой машины — вполне вероятно, необходимо будет поставить задание сотруднику кофейни утилизировать невостребованный напиток);
- POST /coffee-machines/{id}/operations выполняет операции на кофе-машине, сгенерированные на предыдущем шаге.

Обратите также внимание, что содержание операции «отменить заказ» изменяется на каждом из уровней. На пользовательском уровне заказ отменён, когда решены все важные для пользователя вопросы. То, что отменённый заказ какое-то время продолжает исполняться (например, ждёт утилизации) — пользователю неважно. На уровне исполнения же нужно связать оба контекста:

- GET /tasks/{id}/status
 {"status":"canceled","operations":{"status":"canceling","items":[...]}}
 — с т.з. высокоуровневого кода задача завершена (canceled), но с точки зрения низкоуровневого кода список исполняемых операций непуст, т.е. задача продолжает работать.

NB: так как task связывает два разных уровня абстракции, то и статусов у неё два: внешний canceled и внутренний canceling. Мы могли бы опустить второй статус и предложить ориентироваться на содержание operations, но это вновь (а) неявно, (б) предполагает необходимость разбираться в более низкоуровневом интерфейсе operations, что, быть может, разработчику вовсе и не нужно.

Может показаться, что соблюдение правила изоляции уровней абстракции является избыточным и заставляет усложнять интерфейс. И это в действительности так: важно понимать, что никакая гибкость, логичность, читабельность и расширяемость не бывает бесплатной. Можно построить API так, чтобы оно выполняло свою функцию с минимальными накладными расходами, по сути — дать интерфейс к микроконтроллерам кофе-машины. Однако пользоваться им будет крайне неудобно, и расширяемость такого API будет нулевой.

Дублирование функций на каждом уровне абстракций позволяет добиться важной вещи: возможности сменить нижележащие уровни без необходимости переписывать верхнеуровневый код. Мы можем добавить другие виды кофе-машин с принципиально другими физическими способами определения готовности напитка, и наш метод GET /orders?order_id={id} продолжит работать, как работал.

Да, код, который работал с физическим уровнем, придётся переписать. Но, во-первых, это неизбежно: изменение принципов работы физического уровня автоматически означает необходимость переписать код. Во-вторых, такое разделение ставит перед нами четкий вопрос: до какого момента API должно предоставлять публичный доступ? Стоило ли предоставлять пользователю методы физического уровня?

Разграничение областей ответственности

Исходя из описанного в предыдущей главе, мы понимаем, что иерархия абстракций в нашем гипотетическом проекте должна выглядеть примерно так:

- Пользовательский уровень (те сущности, с которыми непосредственно взаимодействует пользователь и сформулированы в понятных для него терминах; например, заказы и виды кофе).
- Физический уровень (непосредственно сами датчики машины).

Теперь нам необходимо определить ответственность каждой сущности: в чём смысл её существования в рамках нашего API, какие действия можно выполнять с самой сущностью, а какие — делегировать другим объектам. Фактически, нам нужно применить «зачем-принцип» к каждой отдельной сущности нашего API.

Для этого нам нужно пройти по нашему API и сформулировать в терминах предметной области, что представляет из себя каждый объект. Напомню, что из концепции уровней абстракции следует, что каждый уровень иерархии — это некоторая собственная промежуточная предметная область, ступенька, по которой мы переходим от описания задачи в терминах одного связываемого контекста («заказанный пользователем лунго») к описанию в терминах второго («задание кофе-машине на выполнение указанной программы»).

В нашем умозрительном примере получится примерно так:

1. Заказ `order` — описывает некоторую логическую единицу взаимодействия с пользователем. Заказ можно:
 - создавать
 - проверять статус
 - получать или отменять
2. Рецепт `recipe` — описывает «идеальную модель» вида кофе, его потребительские свойства. Рецепт в данном контексте для нас неизменяемая сущность, которую можно только просмотреть и выбрать.
3. Кофе-машина `coffee-machine` — модель объекта реального мира. Мы можем:
 - получать статус машины
 - получать и изменять состояние машины

Если внимательно посмотреть на каждый объект, то мы увидим, что, в итоге, каждый объект оказался в смысле своей ответственности составным: `Vehicle` одновременно "знает" и про объект реального мира, и про его виртуальное отображение на карте; карта "знает" свою область картографирования - фактически, область на реальной Земле, которую схематически отображает, - и при этом должна предоставлять некоторый контекст для отображения виртуальных графических фигур, и так далее.

Ничего удивительного в этом, конечно же, нет — поскольку API в целом связывает разные контексты, в нём всегда будут объекты, объединяющие термины разных предметных областей. Наша задача - декомпозировать объекты так, чтобы, с одной стороны, разработчикам было удобно и понятно пользоваться нашей иерархией абстракций, а нам, с другой стороны, было удобно такую архитектуру поддерживать.

Декомпозиция интерфейсов

Если каждый объект представляет собой объединение разнородной ответственности в терминах разных предметных областей, каким образом мы можем добиться эффективного уменьшения связанности объектов между собой? Давайте подумаем, где мы можем "сэкономить" на связях.

Возьмём, например, объект Map. Во взаимодействии с объектом source он выступает как чистый источник географических координат, вся прочая ответственность объекта карты source не касается.

Напротив, оверлею географические координаты ни к чему: он существует в некотором графическом контексте, где оперируют пикселями.

Раз смежным объектам знание о полной функциональности карты не нужно, именно здесь мы и можем убрать лишние связи: потребуем, чтобы карта при взаимодействии с источником данных выступала только как картографический контекст, а при взаимодействии с оверлеем - как чисто графический контекст. Для организации такой абстракции используются интерфейсы.

Определим два интерфейса:

- IGeoContext - предоставляет методы работы с областью картографирования;
- IGraphicalContext - предоставляет контекст рендеринга.

Объект Map в этом случае реализует оба интерфейса, однако связанные сущности работают уже не с конкретным объектом Map, а с некоторой реализацией абстрактного интерфейса, ничего не зная о прочих свойствах этого объекта.

NB. Во многих языках программирования нет поддержки интерфейсов, абстрактных классов и/или множественного наследования. Однако выделять интерфейсы нам это не мешает, поскольку мы всегда можем "договориться", что объект source имеет право пользоваться только вот этим набором свойств и методов. Конечно, контролировать соблюдение этой договоренности в достаточно развесистом API довольно сложно, но, поверьте автору, вполне возможно, тем более, что тестами и/или статическим анализом кода соблюдение договоренностей об интерфейсах можно проверить почти всегда.

Разделение контекстов - не единственная причина, по которой выделение интерфейсов критически важно при проектировании API. Предъявление к входящим параметрам требования только удовлетворять интерфейсу существенно упрощает создание альтернативных реализаций ваших объектов, в том числе в целях тестирования. Теперь чтобы протестировать объект `source` достаточно написать `mock` на `IGeoContext`, а не весь класс `map` целиком. Аналогично, если мы захотим использовать наши оверлеи для показа их, скажем, в качестве какой-то инфографики или на абстрактном плане местности, нам не придётся переделывать для этого класс `Map` - достаточно будет альтернативной реализации `IGraphicalContext`.

При выделении интерфейсов важно также понимать, что интерфейс, в отличие от его реализации, должен быть минимально достаточным и не должен включать в себя вспомогательные методы. Например, если класс `map` имеет как метод для получения всей области картографирования в виде четырехугольника `getBBox`, так и методы получения каждого из углов по отдельности - `getLeftBottom`, `getRightTop`, например, — то интерфейс `IGeoContext` должен содержать что-то одно. Нет никакого смысла загромождать интерфейс альтернативными реализациями одной и той же функциональности — это затрудняет чтение и усложняет написание собственных реализаций. Если только нет каких-то показаний с точки зрения производительности, следует отдать предпочтение максимально общему методу - в нашем случае `getBBox`.

Интерфейсы как универсальный паттерн

Как мы убедились в предыдущей главе, выделение интерфейсов крайне важно с точки зрения удобства написания кода. Однако, интерфейсы играют и другую важную роль в проектировании: они позволяют уложить в голове архитектуру API целиком.

Любой сколько-нибудь крупный API рано или поздно обрастает разнообразной номенклатурой методов, как в силу того, что в одном объекте «сходятся» несколько предметных областей, так и в силу появления со временем разнообразной вспомогательной и дополнительной функциональности. Особенно сложной номенклатура объектов и их методов становится в случае появления альтернативных реализаций одного и того же интерфейса.

Человеческие возможности не безграничны: невозможно держать в голове всю номенклатуру объектов. Это осложняет и проектирование API, и рефакторинг, и просто решение возникающих задач по реализации той или иной бизнес-логики.

Держать же в голове схему взаимодействия интерфейсов гораздо проще - как в силу исключения из рассмотрения разнообразных вспомогательных и специфических методов, так и в силу того, что интерфейсы позволяют отделить существенное (в чем смысл конкретной сущности) от несущественного (деталей реализации).

Поскольку задача выделения интерфейсов есть задача удобного манипулирования сущностями в голове разработчика, мы рекомендуем при проектировании интерфейсов руководствоваться, прежде всего, здравым смыслом: интерфейсы должны быть ровно настолько сложны, насколько это удобно для человеческого восприятия (а лучше даже чуть проще). В простейших случаях это просто означает, что интерфейс должен содержать семь плюс-минус два метода. Более сложные интерфейсы должны декомпозироваться в несколько простых.

Это правило существенно важно не только при проектировании API - не забывайте, что ваши пользователи неизбежно столкнутся с той же проблемой - понять примерную архитектуру вашего API, запомнить, что с чем связано в вашей системе. Правильно выделенные интерфейсы помогут и здесь, причём сразу в двух смыслах - как непосредственно работающему с вашим кодом программисту, так и документатору, которому будет гораздо проще описать структуру вашего API, опираясь на дерево интерфейсов.

С другой стороны надо понимать, что бесплатно ничего не бывает, и выделение интерфейсов - самая «небесплатная» часть процесса разработки API, поскольку в чистом виде приносится в жертву удобство разработки ради построения «правильной» архитектуры: разумеется, код писать куда проще, когда имеешь доступ ко всем объектам API со всей их богатой номенклатурой методов, нежели когда из каждого объекта доступны только пара непосредственно примыкающих интерфейсов, притом с максимально общими методами.

Помимо прочего, это означает, что интерфейсы необходимо выделять там, где это актуально решаемой задаче - прежде всего, в точках будущего роста и там, где возможны альтернативные реализации. Чем проще API, тем меньше нужда в интерфейсах, и наоборот: сложное API требует интерфейсов практически всюду просто для того, чтобы ограничить разрастание излишне сильной связанности и при этом не сойти с ума.

В пределе в сложном API должна сложиться ситуация, при которой все объекты взаимодействуют друг с другом только как интерфейсы — нет ни одной публичной сигнатуры, принимающей конкретный объект, а не его интерфейс. Разумеется, достичь такого уровня абстракции практически невозможно - почти в любой системе есть глобальные объекты, разнообразные технические сущности (имплементации стандартных структур данных, например); наконец, невозможно «спрятать» за интерфейсы системные объекты.

Информационные контексты

При выделении интерфейсов и вообще при проектировании API бывает полезно взглянуть на иерархию абстракций с другой точки зрения, а именно: каким образом информация протекает через нашу иерархию.

Вспомним, что одним из критериев отделения уровней абстракции является переход от структур данных одной предметной области к структурам данных другой. В рамках нашего примера через иерархию наших объектов происходит трансляция данных реального мира - географическое положение и реальные свойства транспортного средства - через добавление высокоуровневых опций (вид иконки этого транспортного средства) в графические примитивы конкретной платформы (svg-оверлей, заданный в пиксельных координатах сцены).

Мы уже отмечали, что одним из недостатков нашей первой "наивной" реализации `api` была необходимость объекту `source` "знать" о том, как осуществляется преобразование данных (геокоординат в пиксели). Если это правило обобщить, оно будет выглядеть следующим образом:


- каждый объект в иерархии абстракций должен оперировать данными согласно своему уровню иерархии;
- Преобразованием данных имеют право заниматься только те объекты, в чьи непосредственные обязанности это входит.

Из этих правил явно следует, что в нашей системе `source` имеет право оперировать только географическими координатами, а оверлей - только пиксельными. Оперировать и теми, и другими имеет право только объект, реализующий оба интерфейса `IGeoContext` и `IGraphicalContext`, то есть `Map`. Аналогично, `source` имеет право оперировать только свойствами реальных транспортных средств (идентификационный номер), оверлей - только свойствами их графического представления (иконка, её размер), и только `Vehicle` может связать идентификатор объекта со свойствами его графического представления.

Достаточно внимательный читатель в этом месте может заметить, что правило информационной иерархии всё равно нарушается: объект высшего уровня `Map` оперирует данными низшего уровня абстракции - пикселями сцены, и будет совершенно прав. Конечно, в нашем умозрительном примере это совершенно излишне, но в реальном "большом" `api` карту от этого знания необходимо избавить.

Для этого необходимо ввести некоторую промежуточную сущность, которая, с одной стороны, будет следить за изменением области картографирования и предоставлять методы трансляции геокоординат в пиксели сцены; с другой - предоставлять графическим примитивам холст для рисования - родительский `svg`-элемент в нашем случае.

Назовем такую сущность, скажем, `IRenderingEngine`; в нашей иерархии это. Интерфейс займёт промежуточное положение между картой и графическими объектами. Тогда ответственностью карты как `IGraphicalContext` будет предоставление доступа к своему `rendering engine`; соответственно, оверлей будет работать уже не с `IGraphicalContext`, а именно с `engine`; связывание же оверлея с его графическим

контекстом должен произвести тот, кто оверлей инстанцирует - в нашей системе это может быть либо  либо vehicle, либо сам графический контекст, либо какая-то третья сущность, которую мы выделим специально для этого.

Дерево информационных контекстов (какой объект обладает какой информацией, и кто является транслятором из одного контекста в другой), по сути, представляет собой "срез" нашего дерева иерархии интерфейсов; выделение такого среза позволяет проще и удобнее удерживать в голове всю архитектуру проекта.

Связывание объектов

Существует множество техник связывания и управления объектами; ряд паттернов проектирования - порождающие, поведенческие, а также MV*-техники посвящены именно этому. Однако, прежде чем говорить о конкретных паттернах, нужно, как и всюду, ответить на вопрос "зачем" - зачем с точки зрения разработки API нам нужно регламентировать связывание объектов? Чего мы хотим добиться?

В разработке программного обеспечения в целом снижение связанности объектов необходимо прежде всего для уменьшения сайд-эффектов, когда изменения в одной части кода могут затронуть работоспособность другой части кода, а также для унификации разработки.

Разумеется, к API эти суждения также применимы с поправкой на то, что изменения происходят не только при рефакторинге кода; API так же должно быть устойчиво:

- к изменениям в реализации других компонентов api, в том числе модификации объектов API сторонними разработчиками, если такая модификация разрешена;
- к изменениям во внешней среде - появлению новой функциональности, обновлению стороннего программного и аппаратного обеспечения, адаптации к новым платформам.

Необходимость слабой связанности объектов API также вытекает из требования дискретных интерфейсов, поскольку поддержание разумно минимальной номенклатуры свойств и методов требует снижения количества связей между объектами. Чем слабее и малочисленнее связи между различными частями API, тем проще заменить одну технологию другой, если возникнет такая необходимость.

Проблема связывания объектов разбивается на две части:

- установление связей между объектами;
- передача сообщений/команд.

Установление связей между объектами

В нашем примере нам нужно установить связи между источниками данных - `map` и `source` - и объектами, эти данные представляющими - `vehicle` и `overlay` - при посредничестве промежуточных сущностей - `renderingEngine`. При этом вариантов, как же нам связать эти объекты друг с другом просматривается множество: фактически, связь должна быть - напрямую или опосредованно - между любой парой объектов.

Можем, например, сделать вот так:

- карта принимает при создании `source` как параметр конструктора;
- карта инстанцирует `vehicle`, хранит список всех созданных объектов и обновляет им координаты;
- разработчик сам создаёт `renderingEngine` и прикрепляет его к карте методом `setRenderingEngine`;
- после прикрепления `engine` карта создаёт оверлеи на каждый объект `vehicle` и передаёт их в `renderingEngine` для отрисовки;
- При смене области просмотра карта вызывает метод `setViewport` у `source`;
- При изменении географических (вследствие обновления) или пиксельных (вследствие смены области просмотра) координат карта перебирает все созданные ей оверлеи и устанавливает им новые координаты.

В этой схеме нарочно допущены все мыслимые ошибки проектирования. Разберём их в порядке, описанном в предыдущих главах, от области применения к конечным интерфейсам.

Во-первых, мы грубо проигнорировали кейсы использования. В нашей схеме у карты только один несменяемый источник данных, хотя разработчику может понадобиться как несколько карт с одним источником, так и множество источников на одной карте. Обратите внимание, кейс "много карт на один источник" мы сами себе заблокировали, заставив карту вызывать `setViewport` источнику - теперь источник не може отличить, какая из нескольких карт изменила область просмотра. При этом, при наличии нескольких карт, один `vehicle`, вполне возможно, будет отображаться сразу на нескольких картах.

Во-вторых, мы переступили через уровень абстракции, заставив карту задавать пиксельные координаты оверлеям. Это приводит к тому, что мы, возможно, будем не в состоянии реализовать дополнительные движки рендеринга и даже оптимизировать старые - например, движок мог бы оптимизировать движение карты на небольшие смещения, отрисовывая графическое окно с запасом и перемещая только область показа, а не все объекты.

В-третьих, мы создали объект-"швейцарский нож" - карту, которая следит за всем и реализует все сценарии, что приводит к сложностям в рефакторинге, тестировании и поддержке этого объекта.

Наконец, в-четвёртых, вместо того, чтобы сделать выбор движка рендеринга автоматизированным, мы заставляем разработчика всегда самому выбирать технологию, что усложняет работу с API, вынуждая разбираться в дополнительных, не связанных с решаемой задачей, концепциях и приводит к невозможности сменить движок по умолчанию в будущем.

Следует заметить, что каждую из этих задач можно закостылять и решить без нарушения обратной совместимости. Например, можно сделать в объекте source метод порождения дочерних объектов, выполняющих тот же интерфейс, чтобы передавать их в конструкторы map - таким образом можно будет привязывать несколько карт к одному источнику. Проблему с выставлением пиксельных координат оверлея можно решить, написав новую реализацию оверлея, которая не будет применять полученные координаты, а обратится в renderingEngine для пересчёта полученных от карты координат в актуальные. И так далее, и тому подобное - через несколько итераций таких "улучшений" мы получим типичное современное API, в котором каждая функция есть магический чёрный ящик, выполняющий что угодно, кроме того, что написано в её названии, а для полноценной работы с таким API нужно прочитать не только всю документацию, но и комментарии на форумах разработчиков относительно неочевидной работы тех или иных функций.

Попробуем теперь спроектировать API правильно. Начнём с отношений map и source.

Мы знаем, исходя из сценариев использования, что связь map и source имеет вид "многие ко многим", хотя кейс "одна карта - один источник" является самым частотным. Мы знаем, что разработчики будут реализовывать свои source, но вряд ли свои map. Мы также знаем, что эти реализации будут сильно различаться по потребностям - в каких-то системах потребуются оптимизация - так, чтобы source следил только за видимыми объектами - а в каких-то, напротив, объектов мало и заниматься оптимизацией преждевременно.

Отсюда мы можем сформулировать наши требования к связыванию:

- начальное привязывание одиночного source к карте должно быть максимально упрощено;
- должны быть методы добавления и удаления связей в runtime;
- стандартные реализации source и map должны максимально упростить реализацию сложного кейса "к одному source подключено много map, и он оптимизирует слежение, запрашивая только видимые объекты"
- разработчик должен иметь возможность реализовать свою имплементацию source, работающего по иным принципам, не прибегая к необходимости костылить методы.

Поскольку и карта, и источник должны знать друг о друге (источник отслеживает изменение области просмотра карты, а карта отслеживает обновление состояния

источника), нам придётся создать парные методы добавления и удаления связей и в карте, и в источнике, что приводит нас к интерфейсу вида:

```
Map.addSource(source) Map.removeSource(source) Source.addMap(map) Source.removeMap(map)
```

Однако такой интерфейс чрезвычайно не очевиден: из него не понятно, что для успешного связывания нужно выполнить и `addSource`, и `addMap`.

Мы можем упростить этот момент, реализовав эти методы так, чтобы они сами выполняли вызов связанного метода. Однако, проблемы разработчика это не решит: всё ещё неясно, каким методом пользоваться правильно.

Популярное решение состоит в том, чтобы объявить один из методов точкой входа для разработчика, а второй объявить техническим и запретить его вызывать иначе как из другого. Расширенным вариантом этого решения является объявление обеих пар методов техническими и создание специального объекта, через который осуществляется связывание.

На самом деле, такое решение не внесёт больше понимания. Из номенклатуры методов неясно, что же конкретно они делают. Попросту скрыть их также нельзя: разработчику, пишущему свою имплементацию `source` или `map`, всё равно придётся эти методы реализовать и разобраться в механизме их работы.

Чтобы выйти из этого порочного круга, вспомним о правиле декомпозиции интерфейсов. Если `source` имеет единственную задачу быть источником данных, то карта - композиция нескольких интерфейсов, и во взаимоотношениях с `source` должна выступать не как объект `map`, а как некий абстрактный провайдер сведений о наблюдаемой области.

Следовательно, метода `addMap` у `source` быть не может - для него добавление карты означает появление дополнительной отслеживаемой области. Метод должен выглядеть примерно следующим образом:

```
Source.addObserver(IGeographicalContext, IObserver)
```

При выполнении этого метода `source` начинает передавать `observer`-у сведения о том, что происходит в указанной области. Тогда мы можем реализовать метод `addSource` так, чтобы он создавал `observer` и привязывал карту-контекст к источнику.

В этом решении ситуация выглядит понятной и логичной и в решении стандартных кейсов, так и при написании собственных имплементаций `source`.

Перейдём теперь к вопросу создания и связывания `vehicle` с картой и источником.