7 Архитектура с эволюционным развитием: ловушки и антипаттерны

Мы потратили много времени, обсуждая надлежащие уровни связанности в архитектуре. Однако мы также живем в реальном мире и наблюдаем множество связанностей, которые *вредят* способности проекта эволюционировать.

Были выявлены две плохие практики проектирования, которые проявляются в проектах программного обеспечения: *ловушки* и *антипаттерны*. Многие разработчики используют слово *антипаттерны* как жаргонный термин «плохой», но его реальное значение требует уточнения. Антипаттерн программного обеспечения состоит из двух частей. Первая часть: антипаттерн является практикой, которая первоначально кажется хорошей идеей, но оборачивается ошибкой. Вторая часть: для большинства антипаттернов существует множество гораздо лучших альтернатив. Разработчики архитектуры замечают много антипаттернов только в ретроспективе, поэтому их тяжело избежать. *Ловушка* на первый взгляд выглядит как хорошая идея, но немедленно проявляет себя как плохой путь. В этой главе мы рассматриваем ловушки и антипаттерны совместно.

Техническая архитектура

В этом разделе акцент делается на распространенную практику, используемую в промышленности, которая особенно вредит способности команды эволюционировать архитектуру.

Антипаттерн: Vendor King

Некоторые крупные предприятия приобретают программное обеспечение Планирование ресурсов предприятия (ERP — Enterprise Resource Planning) для решения распространенных бизнес-задач, таких как ведение учета, управление инвентаризацией и другие рутинные операции. Это работает, если компании готовы сгибать свои бизнес-процессы и другие решения для того, чтобы приспособить этот инструмент, и может быть использовано стратегически, когда разработчики архитектуры понимают ограничения и преимущества.

Однако многие организации становятся чрезмерно амбициозными в отношении программного обеспечения этой категории, что приводит к *антипаттерну король-поставщик* (*vendor king*), архитектура которого целиком строится на основе продукции поставщика, что патологически привязывает организацию к этому инструменту. Компании, приобретающие программное обеспечение поставщика, планируют увеличение пакета с помощью его плагинов, чтобы расширить основную функциональность для приведения ее в соответствие с предметной областью предприятия. Однако продолжительное время инструмент ERP невозможно настроить в достаточной степени для полной реализации того, что необходимо, и разработчики обнаруживают свою беспомощность в результате ограничений инструмента и из-за того, что они сделали его центром архитектурной вселенной. Другими словами, разработчики архитектуры сделали из поставщика короля своей архитектуры, диктующего принятие в будущем решений.

Для того чтобы избежать антипаттерна, следует рассматривать программное обеспечение просто как другую точку интеграции, даже если у него первоначально был широкий круг обязанностей. Предполагая интеграцию на начальном этапе, можно легче менять бесполезные характеристики с другими точками интеграции, свергая короля с престола.

Поместив внешний инструмент или платформу в самое сердце архитектуры, разработчики значительно ограничили свои возможности эволюционировать в двух основных направлениях, а именно технически и с точки зрения бизнес-процесса. Разработчики технически ограничены выбором поставщика в отношении систем хранения данных, поддерживаемой инфраструктурой и массой других ограничений. С точки зрения предметной области крупный инструмент инкапсуляции в конечном счете страдает от антипаттерна «Ловушка на последних 10 %» (с. 206). С точки зрения бизнес-процесса этот инструмент не может поддерживать оптимальным рабочий поток; это — побочный эффект или ловушка на последних 10 %. Большинство компаний завершают работу подчинением платформе, заменяя процессы, а не пытаясь настроить инструмент. Чем больше компаний поступят так, тем меньше отличительных особенностей существует между компаниями, что замечательно, поскольку различие не является преимуществом.

Принцип *давайте остановим работу и назовем это успехом* является одним из тех, которые разработчики обычно учитывают, занимаясь в реальном мире с пакетами ERP. Так как они требуют значительных затрат времени и денежных инвестиций, компании с неохотой на них соглашаются, когда они не работают. Ни один технический отдел не хочет соглашаться на потерю миллионов долларов, а поставщик инструмента не хочет согласиться на плохую многослойную реализацию. Таким образом, каждая из сторон согласна остановить работу и назвать это успехом при большей части нереализованной обещанной функциональности.

Не связывайте свою архитектуру с королем-поставщиком.

Вместо того чтобы стать жертвой антипаттерна короля-поставщика, попробуйте рассматривать продукты поставщика как еще одну точку интеграции. Разработки могут изолировать изменения инструмента поставщика от воздействия их архитектуры, построив между точками интеграции слои противодействия разрушениям.

Ловушка: дырявая абстракция

> Все нетривиальные абстракции в какой-то степени дырявые.
>
> *— Джоэл Спольски (Joel Spolsky)*[1]

Современное программное обеспечение покоится на башне абстракций: операционные системы, платформы, зависимости и т. п. Как разработчики, мы строим абстракции так, что у нас нет возможности постоянно думать, находясь на нижних уровнях. Если разработчикам потребовалось перевести бинарные цифры, которые поступают с жестких дисков в текст для программы, они никогда не смогут что-нибудь сделать! Одним из триумфов современного программного обеспечения является то, как хорошо мы можем строить эффективные абстракции.

Но абстракции дорого обходятся, потому что нет совершенных абстракций, а если бы они были, то это были бы не абстракции, а нечто реальное. Как считает Джоэл Спольски, все нетривиальные абстракции имеют дырку (протекают). Это является проблемой для разработчиков, потому что мы верим, что абстракции всегда точные, но они часто удивительным образом разрушаются.

Повышенная сложность технологических стеков недавно превратила абстракцию в разрушительную проблему. На рис. 7.1 представлен типичный технологический стек, относящийся примерно к 2005 году.

В этом стеке имя поставщика на блоках меняется в зависимости от местных условий. Со временем, по мере того как программное обеспечение становится все в большей степени специализированным, наш технологический стек становится все более сложным, как показано на рис. 7.2.

---

[1] http://russian.joelonsoftware.com/Articles/LeakyAbstractions.html

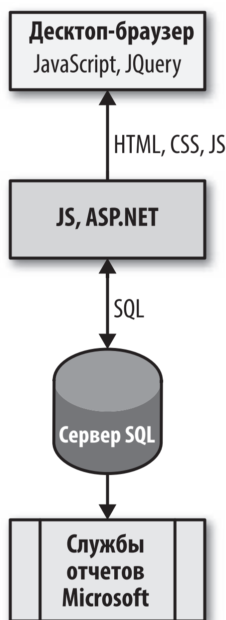

Рис. 7.1. Типичный технологический стек 2005 г.

Как видно на рис. 7.2, каждая из частей экосистемы программного обеспечения расширилась и стала более сложной. По мере того как проблемы, с которыми сталкиваются разработчики, становятся все более сложными, такими же сложными становятся и их решения.

Первоначальная дырявая абстракция, где разрушающая абстракция на низком уровне приводит к неожиданному хаосу, является одним из побочных эффектов повышения сложности технологического стека. Что, если одна из абстракций на самом нижнем уровне проявляет сбой, например, некоторый неожиданный побочный эффект из кажущегося безвредным вызова базы данных? Поскольку существует так много слоев, то этот сбой будет двигаться в верхнюю часть этого стека, возможно, вызывая на своем пути «метастазы», проявляясь в глубоко встроенном сообщении об ошибке в UI. Отладка и ретроспективный анализ становятся тем затруднительнее, чем сложнее технологический стек.

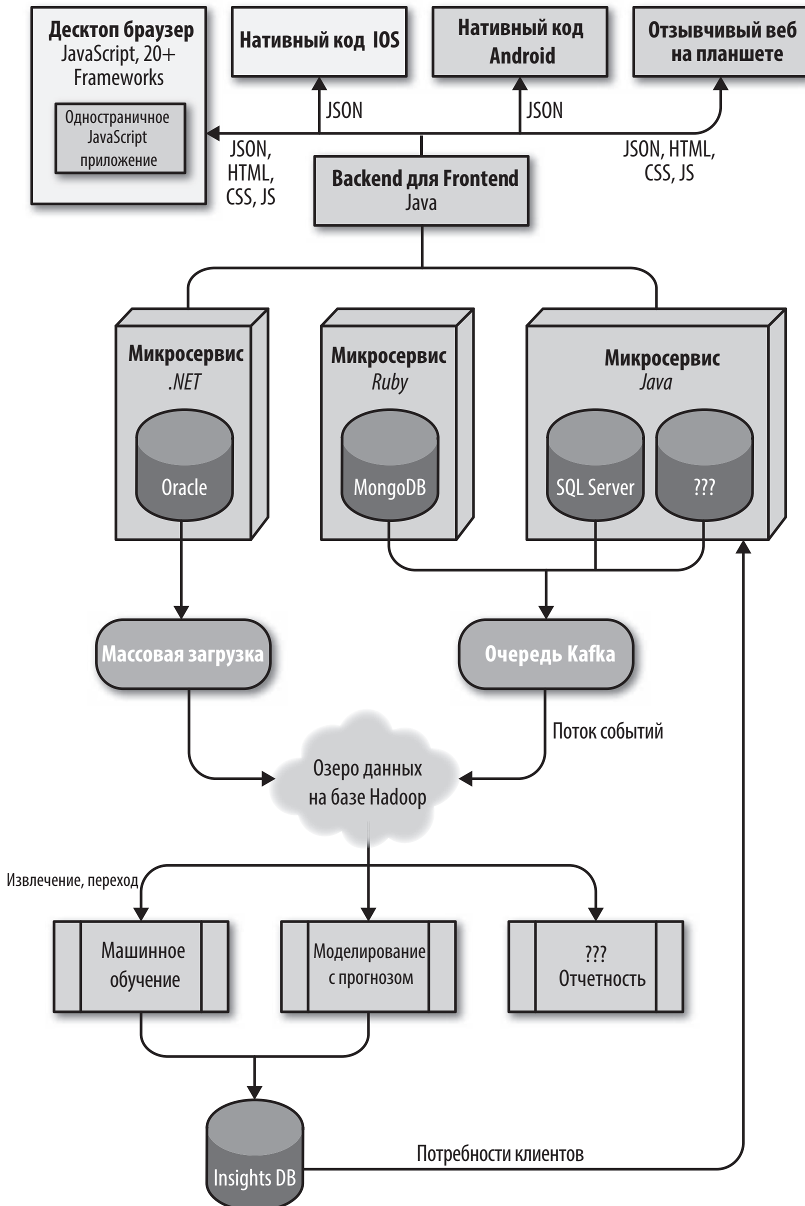

Рис. 7.2. Типичный программно-реализованный стек, относящийся к 2016 году, с множеством движущихся частей