The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Beyond Software Architecture

Creating and Sustaining Winning Solutions

Luke Hohmann

Forewords by Martin Fowler and Guy Kawasaki

Beyond Software Architecture

Transaction-based business models are found almost exclusively within enterprise software. Creative marketects know how to define a transaction and construct a transaction fee that works best for their target market. It is imperative that the tarchitect understand both the legal and the business-model transaction definition.

Sounds Great, But What Do I Charge?

A business model defines *how* you will charge a customer for your products and/or services but not *how much*. A *pricing model* defines how much. Transaction fees, for example, can be just a few cents to many millions of dollars depending on the nature of the transaction! If your business model is based on access to the software, the associated pricing model could be a one-time payment based on the specific modules or features licensed (a "menu" approach), or it could be a set fee based on the annual revenue of the enterprise.

Pricing a product or service is one of the hardest of the marketect's jobs. Charge too much and you face competition, lower revenue (too few customers can afford your product), and slow growth. Charge too little and you leave money on the table. While a detailed discussion of pricing is beyond the scope of this book, here are some principles that have worked well for me.

- Price should reflect value. If your customer is receiving hundreds of thousands of dollars of quantifiable benefits from your product, you should receive tens of thousands of dollars or more. From the perspective of your customer, price isn't affected by what the product costs but by what it's worth.
- Price should reflect effort. If your application requires a total development team of 120 people, including developers, QA, technical publications, support, product management, marketing, and sales, then you need to charge enough to support them. From the perspective of your employer, if you're not going to charge enough to be profitable, your product will be shut down. And it should be. Either it isn't providing enough value or the value it provides costs too much to create.
- Price should support your positioning. If you're positioning yourself as "premium," your price will be high. If you're positioning yourself as "low cost," your price will be low.
- Price must reflect the competitive landscape. If you're the new entry into an established market, a new business model or a new pricing model may win you business. Or it may simply confuse your customers. If it does, you're probably going to end up switching to a business model that your customers understand. Either way, understanding the business models of your competitors will help you make the right initial choice and will help you quickly correct for a poor one.

- Pricing models should not create confusion among your customers. I realize that this can be difficult, especially when you're creating a system with a lot of options. Find ways to reduce the complexity of the menu approach by offering bundles, or just plain simplify your offerings.
- Pricing models should reflect market maturity. If you're in an emerging market, you may need to try several different pricing models before you choose the one that works best. Be careful with your experiments because later customers will ask earlier customers how much they've paid. Note that you can only do this if you have a method for tracking and evaluating the model performance. This can be a tarchitectural issue, in that you may need your software to report certain information to back-office systems when it is installed or used.
- It is usually harder to raise prices than to lower them for the same product. If you start too low and you need to raise prices, you may have to find a way to split and/or modularize your offering.
- Pricing models should reflect your target market. Selling essentially the same solution at different price points to different markets (e.g., the student or small office version) often makes good sense.

Metering

Metering is a business model based on constraining or consuming a well-defined resource or something that the application processes. A constraint model limits access to the system to a specific set of predefined resources. A consumptive model creates a "pool" of possible resources that are consumed. The consumption can be based on concurrency, as when two or more resources simultaneously access or use the system, or on an absolute value that is consumed as the application is used. When all of the available resources are temporarily or permanently consumed, the software becomes inoperable. Many successful business models blend these concepts based on the target market. Here are some of the ways this is done.

Concurrent Resource Management

This business model is based on metering the number of resources concurrently accessing the system, with the most common resource either a *user* or a *session*. The business model is usually designed to constrain the resource ("a license for up to 10 concurrent users"). Both user and session must be defined, because in many systems a single user can have multiple sessions. The specific definition of a resource almost always has tarchitectural implications; managing concurrent users is quite different from managing concurrent sessions, and both are different from managing concurrent threads or processes.

Like transaction fees, concurrent resource business models have a variety of pricing schemes. You may pay less for more resources, and you may pay a different amount for a different resource. Concurrent resource models are almost exclusively the domain of enterprise software.

Identified Resource Management

In this model, specific resources are identified to the application and are allowed to access the system when they have been properly authenticated. The constraint is the defined resources, the most common of which is a *named user*, that is, a specifically identified user allowed to access the application. Identified resource business models are often combined with concurrent (consumptive) resource business models for performance or business reasons. Thus, you may create a business model based on any 10 out of 35 named users concurrently accessing the system or any 3 out of 5 plug-ins concurrently used to extend the application.

The concept of a user as the concurrent or identified resource is so prevalent that marketects should be alert to the ways in which they can organize their business model around users. One common way is to classify users into groups, or types, and separately define the functions and/or applications that each group can access.

The idea is analogous to how car manufacturers bundle optional/add-on features in a car and sell it as a complete package (the utility model versus the sport model). As a result, it is common in concurrent or named user business models to find defined user types (bronze, silver, or gold, *or* standard or professional) with specifically defined functionality associated with each (e.g., a concurrent gold user can access these features . . .).

The administrative burden of this approach can be overwhelming for corporate IT departments. To ease it, try to leverage existing directory or user management infrastructures, such as any AAA (access authentication authorization) or LDAP (Lightweight Directory Access Protocol) servers that may be installed. These systems are designed to assist IT in capturing and managing these data.

Consumptive Resource Management

In this model, you create a specified *amount* of a resource and consume that amount once when the application is invoked or continually while it is running. Unlike a concurrent model, in which consumption varies based on the specific resources simultaneously accessing the system, a purely consumptive model expends resources that are not returned.

Consider time as a consumptive resource. In this approach, you define a period of time (e.g., 100 hours or 30 days) and provide the user with a license for it. As the software is used, it keeps track of the time, "consuming" the designated value from the licenses. When all of the allotted time has been used the software becomes inoperable. Key issues that must be resolved in this approach include the definition of *time* (actual CPU time, system-elapsed time, or other), the manner in which the software will track resource consumption (locally, remotely, or distributed), and the granularity of the time-based license (milliseconds, seconds, days, weeks, months, and so forth).

More generally, it is possible to define an abstract resource and base your business model on metering it. Suppose you have an application for video publishing with two killer features: the ability to automatically correct background noise and the ability to automatically correct background lighting. You could define that any time the user invokes the background noise correction feature they are consuming one computing unit while any time they invoke the background lighting correction feature they are consuming three computing units. You could then provide a license for 20 computing units that the user could *spend* as she deems appropriate.

Consumptive resource models can underlie subscription-based service models—during each billing period (e.g., monthly), for a set fee, you get a predefined number of resources; when the resources are consumed, you can purchase more or stop using the application, and any not consumed are either carried over to the next billing period (possibly with a maximum limit, like vacation days at many companies) or lost forever. This is similar to the access-based subscriptions, except that you are metering and consuming a resource. The difference may be fine-grained, but it is worth exploring the potential benefits of each type because you may be able to access a new market or increase your market share in a given market with the right one. In addition, both models make unique demands on your tarchitecture, so the tarchitect will *have* to know the difference.

Consumptive models have another critical requirement often overlooked—reporting and replenishment. It must be extremely easy for a user/administrator to predict how much an operation will "cost" *before* she decides to spend, the rate at which *spending* is occurring, and when the rate of spending will exceed the allotment for the month or a resource budget is nearing depletion. Because customers will often overspend, it should be painless to buy more. No one will blame you if they run out of a critical resource on Friday afternoon at 6 P.M. Eastern time, just before a critical big push weekend—especially if you warned them yesterday that it would happen. But they will *never*, *ever* forgive you if they can't buy more until Monday at 9 A.M. Pacific time.

Hardware

Hardware-based business models associate the amount charged for the software with some element of hardware. In some cases the software is *free*, but is so intimately tied to the hardware that the hardware is effectively nonfunctional without it. A more traditional approach, and one that is common in business applications, is to associate the business model with the number of CPUs installed in the system. As with all business models, the motivation for this "Per-CPU licensing" is money.

Say an application has been licensed to run on a single machine. If the performance of the machine can be substantially improved simply by adding additional processors, the licensor (software publisher) stands to lose money because the licensee will just add processors without paying any additional license fees! If this same application is licensed on a per-CPU basis, then adding more processors may improve performance but the licensor will get more money for it.

Hardware based business models can be based on any aspect of the hardware that materially affects the performance of the system and can be enforced as required to meet business needs. Per-CPU or *per expansion card* are the most common, but you can also use memory, disk storage (e.g., redundantly mirrored disk drives might be charged twice), and so forth. I wouldn't recommend basing the model on the number of connected keyboards, but you could if you wanted to.

Services

Service-based business models focus on making money from one or more services, not from the software that provides access to them. My favorite example is America Online. AOL doesn't charge for the software; they charge a monthly subscription fee for access to a wide range of services, including e-mail, chat, and content aggregation.

Service-based business models are often used with open source licenses. Examples here include providing assistance in the installation, configuration, and operation of an application or technology licensed as open source or built on open-source software, creating education programs (think O'Reilly) and custom development or integration services.

Creating a service-based business model through open source software (OSS) licensing is a creative approach; however, as of the writing of this book there are no provably sustainable, long-term successful open-source software service business models. This is not an indictment of OSS! I'm aware that most of the Internet runs on OSS, and many companies have very promising service-based business models related to it. I'm merely acknowledging that the market is immature and so such models have yet to be proven. Therefore, any marketect approaching his or her business through OSS should proceed with caution.

Revenue Obtained/Costs Saved

Another business model that is common in enterprise software is a percentage of revenue obtained or costs saved from using the application. Suppose you've created a new CRM (customer relationship management) system that can increase sales to existing customers by an average of 15 percent. You may consider charging 10 percent of the incremental revenue, provided that the total dollar amount is large enough to justify your costs.

Alternatively, let's say that you've created a new kind of inventory tracking and warehouse management system targeted toward small companies ($5M to $50M in annual revenue). Your data indicates that your software will save these companies anywhere from $50K to $1M. A viable business model may charge 15 percent of the savings, again provided that the total dollar amount is sufficiently large.

In choosing these models you have to have a rock-solid analysis that clearly identifies the additional revenues or savings. If you don't, no degree of technical skill in the development team will help the architecture become successful. The fundamental problem is that the data on which these models are based are extremely subjective and easily manipulated. You might think that your new CRM software generated $100,000 more business for Fred's Fish Fry, but Fred thinks it's the spiffy new marketing campaign that Fred, Jr., created. Result? You're not going to get paid the amount you think you've earned.

Enterprises also appear to be more resistant to models based on cost savings. I once tried to create such a model. Even though we had a very solid ROI, the model didn't work and we switched from costs savings to transaction fees. Percentage of revenue obtained or costs saved are also unpopular because they make *customers* track how much they should pay. It is usually much easier to determine how much a customer should pay in the other business models.

Open Source Does Not Mean Free

Sleepycat Software has married a traditional time-based usage business model with an open source certified licensing model in way that drives widespread adoption while still making money. The way it works is that Sleepycat provides its embedded database system, Berkeley DB, as an open source license. The terms allow you to use Berkeley DB at no charge, *provided* that you give away the complete source code for your application under an open source license as well. This is a great example of the viral element of open source licensing.

Many people who use Sleepycat can do this. Proprietary software vendors aren't able to offer their software under the open source license, so Sleepycat sells them a different license for Berkeley DB that permits them to use it without distributing their own source code. This is where the business model comes into play—the way that Sleepycat makes money is very similar to standard licensing models.

Sleepycat can use this approach because the proprietary vendor must link the database into their application and because Sleepycat owns all of the IP associated with the code. The linking process gives Sleepycat leverage to impose licensing constraints, and the ownership of the IP means that Sleepycat can control license terms as they choose. According to Michael Olson, the CEO of Sleepycat, the marriage is working quite well, as Sleepycat has been a profitable company for several years, and is yet another lesson in the benefits of separating your business and license models.