Chapter 4

Business and License Model Symbiosis

Your *business model* is the manner in which you charge customers for your products or services—the way you make money. Associated with every software business model is a license model, which is made up of the terms and conditions (or rights and restrictions) that you grant to a user and/or customer of your software as defined by your business model.

The symbiosis between these models is reflected in how easily we have created shorthand terms that describe them both in the same phrase. "An annual license for 10 concurrent users with the right to receive upgrades and bug fixes" is both a business model based on metered access to the software (by concurrent user) and a license model that defines some of the terms and conditions of its use—the duration, the right to use the software, and the right to receive upgrades and patches. Still, although business and license models are symbiotically related, they are *not* the same thing.

Many software companies practice *Model-T* business and licensing model strategies. They create very few business models (often only one!) and expect each of their target markets to adapt to it. Moreover, they define relatively rigid licensing models, failing to realize that within a business model market segments will pay to obtain certain kinds of rights (like the right to upgrade) or remove certain kinds of restrictions. Rigidly defined business and licensing models may work for a new product in an emerging market, but mature markets need flexibility to capture the maximum revenue and market share from each segment. Just as we want to choose the color of our cars, so customers want choice in how they license their software.

Instead of forcing each target market to adopt a single business or licensing model, it is better to examine each target market to determine the combination that will provide you with the greatest market share and revenue. Doing this well requires that your product marketing organization understand the basics of each model and

how these work in a given target market. Given the plethora of possible models, this can be a challenge.

Considering all of the work involved in creating and supporting a business model, you may wonder why software is licensed and not sold (like a pen or a coffee mug). Perhaps the biggest reason is control. If someone sells you the software like a physical good, they lose control over what you can do with it. Thus, someone selling you software can't prevent you from modifying it, reselling it, reverse-engineering it to determine how it works and using that information to make competing products, or copying it and loading it on multiple computers or making it available over a network. A physical object does not suffer from these problems, which is why nearly all software—even the software you're using to create your next program—is distributed via license.

As one of the key embodiments of the marketecture, the business model and its licensing models may have a *tremendous* impact on your tarchitecture, imposing requirements throughout the system. More sobering is the fact that if your business model is the way you make money, creating the right one, licensing it the right way, and supporting it technically are essential to a truly winning solution. Missing any one of these things, for any given target market, means you might lose it all.

This chapter explores business and license models and the effects they can have on software architecture. I'll review how to make certain your business models are properly supported, discuss key license models often associated with business models, and present a few emerging techniques for making certain you're getting the most from your model. When I'm finished, you'll have the background you need to create effective business and licensing models of your own.

Business Model Nirvana

A key objective of the marketect is to ensure that the company is profitably rewarded for the value it provides to its customers. This often requires multiple co-existing business models because customers often perceive different kinds of value with different functions of a software system. In one application I worked on, the main server was licensed to customers based on transaction fees while optional modules needed for large installations were annually licensed. This combination of business models matched how our customer received value from the system and allowed us to sell to a larger total market. Transaction fees have the virtue of scaling with size in that small customers, having few transactions, pay less while larger customers, having many transactions, pay more. Thus, only customers that need them license the optional modules. Careful design of tarchitecture to support these two models was key to creating a winning solution.

Hohmann_ch04.fm Page 69 Thursday, November 6, 2003 3:18 PM

Common Software Business Models

The most common software-related business models make money by

- Providing unfettered *access* to or *use* of the application for a defined period of time
- Charging a percentage of the *revenue obtained* or *costs saved* from using the application
- Charging for a *transaction*, a defined and measurable unit of work
- *Metering* access to or use of the application, or something the application processes
- Charging for the *hardware* the application runs on or hardware intimately associated with the application, and not the application itself
- Providing one or more *services* that are intimately related to application operation and/or use

Traditional software publishers or independent software vendors (ISVs) rely heavily on the first four models. The other models may be associated with an ISV or may be based on an open-source licensing model. For example, you can't *sell* Apache, but you can certainly base your business model on installing, configuring, and operating Apache-based servers.

Before you read further, go back to the beginning of this list and replace the word *application* with *feature*. Applications are collections of features, each of which can be offered under a different business and licensing model. Thus, for example, you can provide unfettered access to the base application for an annual license fee but charge a

Will That Be an Application, Suite, Bundle, or Feature?

Companies that offer more than one product often try to create synergies around their various product lines. One way of doing this is to associate a set of products as a suite or studio. To be effective, the products the suite comprises should work together and should address some aspect of a common problem domain. Organizationally, it is best if there is a separate marketect in charge of the suite, whose primary job is to work with the marketects of each suite product to make certain the suite is a good value for customers.

A bundle is a simpler concept than a suite and is usually based on a set of products that may have no special relationship to each other, may not interoperate, and may even be normally targeted at different market segments. There may be a separate marketect in charge of a bundle. Ultimately, the marketect must answer the questions associated with creating a suite or a bundle, or with marketing a feature or an application.

transaction fee for invoking a special feature that produces a defined and measurable unit of work. Defining this mix is a key responsibility of the marketect.

The more sophisticated business models are usually associated with enterprise applications, although this is continually changing as the technology to enforce license models matures. Metering is a good example. Many enterprise applications meter by concurrent user. In consumer applications, you might want to meter by the amount of time the user has been accessing your application, but this requires sophisticated technologies, including a reliable way to capture usage data. As these technologies mature, you will find these business models offered wherever they increase business!

Let's consider each of these business models in greater detail, keeping in mind the following points:

- A complex system may have multiple business models. The "base" system might be transaction-fee based while additional "optional" modules might be annually licensed. The central business model should be congruent with the value received from the customer by the generic and expected product. The augmented product may benefit from different licensing models.
- Business models are often coupled with one or more field-of-use license restrictions, which constrain where, when, and how the software can be used. Common field-of-use restrictions include single computer, a single site (called a *site license*), or one or more geographies (which export restrictions may require).
- Field-of-use restrictions work in conjunction with the core business model to create more total revenue for a company. Consider an annual license that is restricted to a single computer. If the licensee wishes to use the software on another computer, he must obtain an additional license, thereby driving more revenue. Field-of-use restrictions are another aspect of the licensing model, much like the "right to upgrade" or the "right to receive technical support." For consistency in contract language and in enforcing your licensing model, it may help to convert a field-of-use restriction to a right (or *entitlement*)—you have the right to use this software on a designated computer, or you have the right to use this software on any computer in your company.
- All of the business models associate some period of time with their use. This period is defined in the license model. Common time periods include months, quarters, and years.
- The licensing model may also define how and when a customer must make payments as a way of extending a given business model to a new market. For example, a service based on a monthly payment originally created for the small to medium business (SMB) market and requiring a purchase order might have to be augmented with the ability to accept credit cards to reach the small office/home office (SOHO) market.

Time-Based Access or Usage

In this model, the licensee can use the software for a well-defined period of time, such as when you *purchase* an operating system. What is really happening is that the operating system publisher, or its agent, has granted you the right to use the operating system, typically on one computer, for that time. Other rights and restrictions will be defined in the license model, but the core business model is based on accessing or using the software (you *pay* to license the software so that you can *use* it).

Common time periods for time-based access or usage include the following.

Perpetual

The licensee is granted a license to use the software in perpetuity—that is, forever. Upgrades or updates are not usually included and instead are separately priced. Bug fixes or patches may be included as part of the original fee, or you may be required to pay a maintenance fee (typically ranging from 15 percent to 30 percent of the initial license fee). Be very careful of perpetual licenses, as they often increase total support and maintenance costs (unless you're careful, you may have to support *every* version—*forever!*).

Despite the potential drawbacks of perpetual licenses, they are surprisingly common and often required in many markets. Consumer software; common productivity tools, such as word processors or presentation software; operating systems; and many enterprise applications are based on them. Perpetual licenses may be required if you're providing a system or technology that may be embedded within other systems. Examples range from the runtime libraries provided by programming language vendors (e.g., the C runtime library) to systems that may become components of hardware devices or information appliances.

Annual

The software can be accessed or used for one year from the effective date of the license. The effective date can be defined as the time the license is granted, the time the software is installed, or the time the software is first used. Annual licenses often include software updates or patches. They may or may not include other services, such as technical support or consulting fees, and they are usually renewable. Renewal may be automatic, and it may be priced differently from the original annual license.

The difference between a perpetual license with an annual maintenance fee and an annual license is subtle but important. In the case of a perpetual license, the maintenance fees are added on as a way of enticing the licensee to pay for additional services, such as bug fixes or new releases. In the case of an annual license, if the licensee hasn't paid and continues to use the software, it is in breach of the license. You will ultimately have to decide how you want to enforce license terms and conditions, a topic explored in greater detail later in this chapter. Annual licenses are common in enterprise-class systems.

Hohmann_ch04.fm Page 72 Thursday, November 6, 2003 3:21 PM

Although perpetual and annual licenses are the most common predefined time periods, there is nothing that prevents you from defining other periods (e.g., nine months) or a collection of times when the software can be accessed (e.g., Monday through Friday from 8 to 5). The hard part is defining one that works best for your target market and making certain you can appropriately enforce the business model and license terms. After all, which market segment would find a nine-month period or only accessing software Monday through Friday from 8 to 5 sensible?

Here are a few additional time-based usage models.

Rental

A rental is a time-based model in which the time allowed for use is set when the license is granted. In reality, it is just an annual license with a different term (or an annual license is just a rental of one year). Rentals are becoming increasingly popular in certain industries, including software test automation and electronic design automation (EDA). As license management technology matures, I predict that rentals will reach all market segments and that we will be able to rent just about every application available.

The business motivations for rental are compelling. For one, a rental allows you to reach a new market. Suppose, for example, that you rent a high-end, professional video editing system for $50,000 for a single-user annual license. One way to reach the home user market is to create a simpler version of this system, with fewer features. Another way is to offer the system as a rental (say, $100 per hour). For the user who has learned the system at work but wants to use it at home, the rental price might be a real bargain—she already knows how to use the application, so why go with anything else?

Key issues that must be resolved with a rental model include specifying when the rental period begins (when you purchase the license? install the software? start using the software?); determining how the software should respond when the rental is finished (will the program stop working entirely, or will you give the user some kind of grace period so that she can save her work?); and pricing (if you can obtain a perpetual license for a graphic design package for $2,995, what is the cost for a one-hour rental? a one-day rental?).

Subscriptions

At the other end of the spectrum is a subscription, in which customers pay a periodic fee to access the software. A subscription is like a rental or an annual license—all that differs is the terms and rights. For example, a rental may not include the right to receive upgrades or patches. Most subscriptions and annual licenses do include this. A rental and a subscription may include technical support, while in many cases you must purchase an additional contract (which can define varying kinds or levels of support) with an annual license. Because subscriptions are often associated with some backend service, the license models that define them are relatively easy to enforce. Simply cut off access to the backend service!

Pay after Use

An interesting question arises in any time-based usage model: What happens when the user accesses the application after the approved period of time? How you answer this question can make or break your relationship with your customer—and your bank account. At one end of the spectrum is absolute enforcement of a business model with no grace period: When the term is finished, the software becomes inoperable. This very severe choice is simply inappropriate for most target markets. An emerging business model charges the customer *after* he has used the software by keeping track of how long he has used it. This model places a host of complex demands on the tarchitecture (see below), but it does help ensure that the company is receiving every dollar it can from the use of its software.

An analogy is a car rental. You're typically charged for each day you rent the car, with strict definitions as to what constitutes a day. If you go beyond the original contract, the car company continues to charge you (they have your credit card on file, and the rate that you'll be charged is predefined in the contract). This pay-after-use business model will become more popular as the relationship between business models, licensing models, and billing systems matures, and as marketects learn to use these new capabilities to create compelling "pay-after-use" solutions.

Per-User Licenses

A special category of software that uses time-based access or usage licenses is applications designed to work on a single personal computer, workstation, or handheld device. As a group, these licenses are often referred to as *per-user*, and often cost less than $500. Per-user licenses work great when the publisher is licensing one copy to one user but not so well when a publisher is trying to sell 500 or 5,000 licenses to an enterprise.

To address this issue, most major software publishers have created *volume licensing* programs to more effectively reach enterprises (businesses, charitable organizations, academic institutions) that license more than one copy of the same software. Volume licensing programs are not fundamentally new but, instead are sophisticated pricing models based on an access- or usage-based business model. They offer enterprises a way to acquire multiple licenses of a given program, usually at a discount. The more copies licensed, the greater the discount.

In a *transactional volume licensing* program, each product is associated with a certain number of *points*. A high-end, expensive product may be worth five points; a low-end, inexpensive product, one point. As an enterprise creates a purchase order (500 licenses to this software, 230 licenses to that software) a point total is dynamically computed. Applicable discounts are calculated based on the total points as specified by the transaction.

In a *contractual volume licensing* program, the enterprise estimates the number of licenses they will need for a projected period of time and commits to acquiring *at least* these licenses by the time specified in the contract. The commitment to purchase a minimum number of licenses entitles the enterprise to a discount. Bonus discounts

may be allowed if additional licenses are acquired, and penalties may be assessed if the customer does not license enough.

Both transactional and contractual licensing programs are highly regulated by the companies that offer them. Consider them whenever you sell multiple "copies" of the same software to one enterprise. Such licenses are more appropriate for applications that can be used in the enterprise context (games are probably not a good choice for a volume licensing program). You may not want to offer a personal finance program as part of a volume licensing deal, largely because the target market will be a single user on a single computer. That said, marketects should be aware of the benefits that a volume licensing program can provide to their customers and their product lines.

OEM

Another category in which time-based access or usage is dominant is in the OEM (original equipment manufacturer) or royalty market. The business model is access to the software. The pricing model is whatever makes sense (in the case of an OEM) or a fairly defined royalty.

Transaction

Transactions are defined and measurable units of work. Business models based on them associate a specific fee with each transaction or a block of transactions. A transaction can be surprisingly simple or maddeningly complex. For example, it can mean *executing* the software. This is common in a business model known as "try and die," in which you can execute the software a predefined number of times—say, five—before it becomes inoperable. I've worked on systems in which a transaction was distributed among multiple servers; the business model was based on charging the customer whose "root" server initiated the transaction.

Fees may be calculated in many different ways. I've worked on systems based on *flat* fees (a fixed cost per transaction), *percentage* fees (a percentage of some other calculated or defined amount), *sliding* fees (the cost per transaction decreases as certain volumes are realized), or *processing* fees, in which the work to perform the transaction is measured and the customer is billed accordingly (e.g., a simple transaction that could be computed with few resources costs less than a complex transaction that required many resources).

There is nothing about a transaction-based model that requires all transactions to be the same size, duration, or amount. Indeed, such differences are the foundation of many business models, and you can use them to create different pricing structures. For example, in the telecommunications industry a phone call is a transaction, the duration of which determines the price.[1]

1. I'm intentionally simplifying this example to illustrate a point. In reality, the price of a call is determined by many complex factors, including, but not limited to, tariffs and the time of day of the phone call. The telecommunications industry is a great case study of how both the marketect and a tarchitect must know the subtleties of the legal issues in creating the business model.

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.

Hohmann_ch04.fm Page 81 Thursday, November 6, 2003 3:23 PM

Rights Associated with Business Models

Marketects choose business models to maximize the benefits to their business. Associated with each of the business models just described is a set of rights and restrictions—things you *can* do or things you *get* and things you *cannot* do or things you *do not get*. I've covered a few of these already, such as the right to use, the right to upgrade, and you can only run this software on one computer.

Your business model should distinguish as many rights and restrictions as possible, because each right is a way of capturing and/or extracting value from your customer and each restriction is a way of protecting your interests. Many business models, for reasons of convenience, competitive advantage, or common practice, often create a *standard* licensing model that blends a variety of rights and restrictions into a single package. Remember, however, that separating them can create greater value.

For example, an annual license to use the software (the business model is time-based access) commonly includes the right to use and the right to receive upgrades but not the right to receive technical support. A subscription to a software-based services model (such as America Online, where the business model is service or metering depending on customer choices) may include all of these rights. This section reviews some of the rights commonly associated with various business models.

Figure 4-1 outlines some the rights and restrictions associated with particular business models. The columns list specific rights or restrictions; the rows are the business models. A check means that this right is commonly allowed by the business model. A number means that the right *may* be allowed by the business model and will be discussed in greater detail. I've added a column that addresses whether or not the fees paid by the licensee are one time or periodic. The timing of fees can affect the choice of model.

The table's focus is the subset of license models most closely correlated with various business models. It does not attempt to capture every possible legal issue that can be defined in a license: exclusivity, warranties, indemnifications, intellectual property rights, confidentiality, or any number of other things that a savvy lawyer can think of. Most of these are not a function of the model but of larger corporate policy that governs every license model, regardless of the business model.

1. Do associate this right with the business model if providing it will give you a competitive advantage, reduce technical or product support costs, or create a stronger tie between you and your customer. Don't if your customers don't care about it or if doing so may cause them more bother or pain that it is worth. For example, in the anti-virus market you have to provide upgrades and bug fixes. In enterprise-class software, upgrades may not be as meaningful because of the large time delays that may exist between release cycles and the often substantial costs associated with integrating the upgrade into the current operational environment.

Business Model	Right to upgrade (to latest version)	Right to receive bug fixes and patches	Right to return	Right to move to a different machine	Right to embed	Right to modify	Right to resell	Support options (e.g., phone, Web, e-mail)	Predefined installation/customization support	One time (1T) or periodic fee (P)
Time-based access				1	1	1	1	1	1	
Perpetual license										1T
Annual license	✓	✓								P
Rental	1	1								1T
Subscription	1	1								P
Pay after use										1T
Transaction	2	2		1	1	1	1	1	1	P
Metering				1	1	1	1	1	1	
Concurrent resource	✓	✓								1T
Identified resource	✓	✓								1T
Consumptive resource	1	1								1T
Hardware	3	3		1	1	1	1	1	1	1T
Service	✓	✓		1	1	1	1	1	1	P

FIGURE 4-1 License rights and business models

2. I'm usually surprised to find that transaction-based business models don't automatically include these rights. In these models, your goal is to drive transactions. Improvements to your software, especially those that can drive transactions, should always be made available to your customers.
3. Do associate these rights if the benefits of point 1 apply and your customers find it relatively easy to apply the upgrade and/or patches. Note that in some hardware-based business models you *don't* want your customers to upgrade to new software. Instead, you want them to purchase entirely new hardware, which is another reason to withhold these rights.

Tarchitectural Support for the Business Model

If your total offering is based on a combination of business models, check for key interactions between the various elements. Review these factors every time the business model changes, because these changes may invalidate prior assumptions and/or choices and motivate one or more changes to the tarchitecture.

Hohmann_ch04.fm Page 83 Thursday, November 6, 2003 3:25 PM

General Issues

The following sections describe the general issues that are present for every business model.

Capturing the Necessary Data

Assess your business model to ensure that your system is capturing all of the data required to make it *work*. Here are two primary categories of data capture:

- *Direct*: The system captures and manages all data necessary to support the business model. It is self-contained.
- *Indirect*: The system must be integrated with one or more other systems to create a complete picture of the necessary data.

To illustrate, a transaction or metered business model will either capture all of the necessary data or work with other systems to capture it. A service-based business model often has to be integrated with other systems, such as payment processing or client management systems, to capture the full set of data needed to create a viable business model.

Reporting/Remittance Requirements

Somewhere along the line your accounting department is going to require that information be submitted for billing purposes. Your job is a lot easier if you can define the format, security, and audibility requirements of these reports.

Business Model Enforcement

What happens when the license to use the software is violated? Does it stop working? Should an entry be made in a log file? Is an e-mail sent to a key person? Should a new billing and licensing cycle be automatically initiated?

Focusing *ility* Efforts

In an ideal world, the tarchitect's ility efforts (reliability, stability, scalability, supportability, usability, and so forth) are congruent with the key objectives of the business model. Suppose your business model is based on processing various transactions, such as check clearing and/or processing in financial services or in health care claims processing for large insurance carriers. In these cases, reliably and accurately computing a lot of transactions quickly is critical to your success. Performance *matters*, and *faster really is better.* Who cares if the product is a bit hard to install or upgrade as long as it's the fastest possible? If your tarchitect and the development organization with building the system also care about performance, you're doing great. If not, you may have some problems.

Of course, performance is not the primary criterion of every business model. In a service-based business model, great performance with lousy customer support may

not be good enough to succeed. Instead of focusing on performance, your development team may need to focus on service elements such as ease of installation and upgrade. A key goal in a service-based business model is to reduce and/or eliminate phone calls to technical support, so it is best to find a tarchitect and a development team who care about these issues. Tradeoffs are determined by how the software will typically be used—applications used infrequently should be easy to learn, while those part of a daily routine should be fast and easy to use.

Increased Revenues or Decreased Costs

Once you know your business model it is easy to focus tarchitectural efforts on activities that increase revenues. Don't forget the *costs* your tarchitecture imposes on your customer. Any time you can reduce those costs, you have more revenue potential. You can reduce costs by making installation and/or upgrade easier or by improving performance and/or backward compatibility so that your users won't have to purchase expensive new hardware every time they upgrade to a new version.

Copy Protection/Antipiracy Protection

All software is subject to illegal copying and distribution, although certain kinds of software, such as that inside a video game cartridge or cell phone, are more resistant to piracy because of the hardware dependency. Even so, hardware is not a strong deterrent—just check out the number of places where you can find replacement software that can boost the performance of your car! Most software, however, is trivially copied, as proven by the many online services that provide easy access to pirated software. Depending on the amount of money you may lose from piracy (some companies estimate several million in losses) you should consider a copy protection scheme, as described further below.

Verifying Business Model and License Model Parameters

Many of the business and licensing models involve setting one or more parameters. For example, an annual license has a definite end date, a concurrent user license has a specific number of concurrent users, possibly further divided by type, and a consumptive, time-based license has the amount of time available for use. Whenever you set a value that is important to your business model, you have to understand when, how, and where this value is set, who can change it, and how its value can be verified. These are nontrivial matters, discussed later in this chapter under enforcing business and licensing models (see also Chapter 16, on Security).

Time-Based Access or Usage

Time-based access or usage business models make few special demands on the tarchitecture, unless you're going to strictly enforce the business model. If this is the case, you're going to need a way of being able to disable the software when the time allotted for use has expired. Many subscriptions don't actually stop a program from working if

you fail to remain current in your payments—you just don't get updates (which has the effect of converting a subscription to a perpetual license).

Transaction

Consider the following when dealing with a transaction-based business model.

Define the Transaction

The definition of a transaction, or the unit of work that is the foundation of the business model, must be clear and unambiguous. Once you've defined the transaction, make certain that the tarchitecture can support it and that it is clearly stated in the software license. This is not easy, but it is essential. As you define the transaction, consider the role that each component is playing with respect to it. In distributed transaction systems, the manner in which an element participates in the transaction must be defined, including which participants are the basis of the business model.

Define the Relationship between the Transaction and the Business Model

I've heard some people say that the first step in starting a phone company is purchasing billing software. While this may not be true, the complexity of the plans offered by cell phones for completed transactions requires sophisticated billing management systems. More generally, it is absolutely essential that you can map the completed transaction to the business model. In the case of the phone company, where the transaction is a phone call, key data include who originated the call, who received it, and how long it took.

Keep Audit Trails

Many transaction-based business models require audit trails that can be used to prove/disprove specific charges. This can be especially vital when attempting to reconcile differences between participants in the transaction. You may also need to cross-reference the transactions created by your system with those created by other systems.

Make Certain that Transactions Are Uniquely Identified

Transactions must be uniquely identified. Be wary of simple database counters, which often won't work in today's distributed environments. Instead of relying on the database vendor, create a truly unique identifier through a reliable algorithm. I've had good luck with the built-in algorithms for creating universally unique identifiers (UUIDs) in UNIX-based systems or the globally unique identifiers (GUIDs) available on MS-Windows systems. Admittedly, these require a lot of space, and represent an extraordinarily large number of unique identifiers. Chances are good you don't need an identifier to be quite that long, and you may be able to safely shorten your unique identifier. Short identifiers that work well are based on alphanumeric codes, like "XR349" or "QPCAZZ." One advantage to these codes is that they are short enough to be used in phone-based self service.

Understand Transaction State, Lifecycle, and Duration

When I'm standing in line at the checkout counter about to make some purchase, my mind often leaps to images of mainframe computers processing hundreds of credit card transactions per second. Of course, there are myriad other kinds of transactions, many of which have complex states, lifecycles, and durations. Managing the duration of the transaction can be particularly complex, especially when a transaction can *live* over a system upgrade. The complete transaction lifecycle must be defined because it impacts your back-office systems that account for transactions.

Suppose, for example, that a transaction can last as long as one year and you bill your customer monthly. Normally, you bill for a transaction when it has completed. However, in this case you could be waiting a long time for money, negatively impacting your cash flow. If you know that 80 percent of transactions complete successfully, you could safely bill when the transaction is started, *provided* that your system can recognize a failed or aborted transaction and your back-office systems can properly adjust and/or otherwise credit your customer's account. Before actually doing this, check with your accounting department to make certain that you're not violating any revenue recognition regulations.

Metering

Metering business models entail many interesting challenges, especially when the metering is based on a concurrent or named user. The following sections address some of the issues that are common to both models.

How Do You Authenticate Users?

Authentication attempts to answer the question "Are you who you say you are?" There are many authentication approaches, from simple user names and passwords to simple-to-use but hard-to-defeat tokens to advanced (and costly) biometric systems. If your business model derives a lot of money from uniquely identifying users, you should consider employing any number schemes beyond simple user names and passwords. More generally, you should work with other, established infrastructure technologies to authenticate users (such as LDAP). Authentication is discussed in greater detail in Chapter 16.

How Many Users?

In a concurrent user system there is some limit to the number of users who can concurrently access it. The manner in which you specify that number is subject to considerable variation. I've worked on systems that span the gamut of managing this value. On the low end, we specified the number of concurrent users as a plain text entry in an `INI` file—completely insecure but entirely acceptable for our requirements. On the high end, we specified the number of concurrent users in a signed license that was then stored on a hardware device using advanced cryptography. To the best of our knowledge, this approach has yet to be cracked!

Hohmann_ch04.fm Page 87 Thursday, November 6, 2003 3:27 PM

How Are You Going to Count Concurrent Users?

While you can, and often should, rely on other parts of the infrastructure to authenticate users, you may not be able to rely on them to count users. This may be a function of your application, or it may be obtained by integrating a license manager into your application.

Are Users Gone or Inactive?

Session management is a key issue in concurrent and named user applications. Once a user is logged in, you can't assume that she will explicitly log out. Something might go wrong: Her client might crash; or she might not remember to log out. The general solution is to associate a timeout parameter that forcibly logs the user off or drops her session after a set period of inactivity.

Unfortunately, setting this value isn't trivial. It must be tuned on the basis of how your application is actually used. Consider an interactive voice system that is accessed by a cell phone. Is a period of inactivity caused by the user pausing to consider his next action or has the line dropped because he drove into a "dead zone"? If the user did drive into a dead zone, how will you reestablish the session so that he doesn't have to re-enter a long string of menu commands? It is important that you provide plenty of configuration parameters on these values so that your administrator can properly tune your application.

Consumptive resource management places severe restrictions on your tarchitecture, primarily because you have to maintain some kind of state. If your customer has purchased "100 hours" of total use, you need to record how long she has actually used the application. One approach is to store these data on a centralized server, which can work if you structure it so you can't be spoofed and you always have a reliable network connection. Another approach is to store usage locally. Be careful, however: It is often trivially easy to reset usage data by erasing previously stored values.

Hardware

The biggest issues with hardware-based models are defining what constitutes the hardware and how you're going to manage it relative to the business model. What, exactly, is a central processing unit (CPU)? In many ways the concept of a CPU is meaningless: Many personal computers and engineering workstations now come with at least two processing units as part of their standard equipment, and some new chip designs have multiple processing units in a single package. Arbitrarily picking one as *central* seems a bit silly. Still, per-CPU business models are common. To support them, you will have to define a CPU and how it will be enforced in your system.

Enforcing Licensing Models

Once you've defined your business and licensing models, and ensured that your tarchitecture can support them, you have to decide how strongly you wish to enforce

the licensing model. Typically, *enforcement* means disabling some or all of the application when the license manager determines that the license has been violated. You can avoid creating or using a license manager and rely on the honor system, create your own licensing manager, or license one from a third-party provider.

The Honor System

The honor system is the simplest and easiest way to enforce a license model. You simply expect your customers to honor the terms of the license, which means not changing the license terms, illegally copying the software, changing configuration parameters to obtain more use of the software, and so forth. You're not giving up any rights, as you still have legal protection under contract law if you find your customer cheating. Rather, you are putting nothing into your application to explicitly prevent your customer from cheating.

I suspect that a large percentage of software publishers rely on the honor system. Whether or not this is a good choice should be based on a careful analysis of several factors, including the relationship you have or want to create with your customer, the potential for lost revenue, and your ability to track the use of your software via the honor system. Compare these against the cost of creating, maintaining, and supporting a more advanced licensing system or licensing such a system from a third party. An additional motivation to use the honor system, especially with enterprise software systems, is that it provides account managers with the opportunity to talk with customers each time the license is up for renewal, and to possibly convince them to purchase more software. In consumer software, where you may have tens of thousands to tens of millions of copies, the honor system may not be a good choice. In enterprise-class software, where you may have several dozen to a few hundred customers, it may be a good choice, especially if you maintain a close relationship with your customers through support or service organizations.

Home-Grown License Managers

In this approach the development team creates the infrastructure for license management. Such a solution is not industrial strength, in that it is often relatively easily defeated. However, it has the advantages of being low cost, lightweight, easy to implement, and completely in control of the development organization. Once you've made the choice to enforce the licensing models, the economics of defeated enforcement must be considered to determine if a home-grown solution is sufficient or a professional system is required. If your software costs $700 per license, a home-grown system may be acceptable. If you're software costs $70,000 per license, there is simply too much money to lose with a home-grown license manager, and you're better off switching to a professional solution.

Some kind of home-grown license management is almost always required for session-based licensing schemes, because the development organization needs complete control over the software's response to the system running out of the internal

It Doesn't Have to Be Unbreakable

One enterprise-class system I helped create provided for a combination of named and concurrent users: Any number of users up to the concurrent user limit could log on at the same time, but only users explicitly identified were allowed access. We implemented our own lightweight but extremely effective licensing manager. The number of concurrent and named users were simply stored in an INI file, with user IDs and passwords stored in a database. Changing either of these entries defeated the licensing scheme, in that if you licensed 20 concurrent users but changed this to 100, you cheated my employer out of 80 concurrent user licenses and tens of thousands of dollars in license fees. We didn't expect anyone to actually cheat, and to this day I know of no company that did. The net was that for a reasonable development effort we were able to devise a scheme that "kept honest people honest."

resources it needs to manage sessions. In other words, do you stop the application (extreme, not recommended) or simply refuse the session (common, but the associated issues are application specific).

Third-Party or Professional License Managers

Professional license managers are typically organized in three main components: the license generator, a client (or local license manager), and a server (or remote license manager). The business model and the software being enforced will determine how you use these components.

The License Generator

The license generator generates a valid license for consumption by the client and/or server. Most license managers will generate digitally signed licenses based on public key cryptography that cannot be altered and that can be used to prove that the rights delivered to the license manager are correct. License generators are typically located at the independent software vendor (ISV) or its agent and are integrated with other backend infrastructure systems, such as order fulfillment, which may initiate license generation, and accounting systems, which may use the records maintained by the license generator for billing. Once a license is generated it can be distributed in a variety of ways, including fax, phone, e-mail, or direct connection from the server to the license generator via the internet.

The Client

The client license manager manages the software running on an end user's computer or workstation. At its simplest terms, it either allows or prevents access to a given

application or licensed feature. The client can typically be configured to operate in one of two ways. One is as a standalone, in which it manages enforcement without working with the server. This mode of operation is common for single-user, consumer-oriented software run on one machine and works well for time-based access or usage models.

Here is an example. You work for SuperSoft, which markets SuperDraw. You want to distribute a 30-day trial of SuperDraw, which can be thought of as a free rental. When SuperDraw is installed the client-side license manager is also installed to enforce these rights along with the digitally signed trial license (e.g., the trial expires and the software cannot be used after 30 days).

The other mode of operation requires the client license manager to work with a server to enforce license rights. This mode is common in business-oriented software that uses metering, such as named or concurrent users. When the application is invoked, the client license manager checks with the server to determine if access is allowed. Your customer must be connected to a network, and you need to consider application behavior when your customer is not connected.

The Server

The server component interprets digitally signed licenses and provides a variety of enforcement services to client license managers. As just described, the server component is required (in one form or another) for the licensing models that support counting or metering.

Here is an example to illustrate how a server works with a client to enforce license rights. You're licensing a high-end CAD system that wants to sell concurrent user licenses at $8K per user. In this case the software can be installed on any available workstation, but each user consumes a concurrent user. The server works with the client to keep track of each concurrent user and ensure that the license terms are properly enforced. This model may seem like a great model, but remember that you still must handle the unconnected user. The easiest way to handle unconnected users is to simply not support them and instead require connected use.

Although I've talked about enforcement, I haven't addressed how your software interoperates with the client and/or server components of the license management system. *Normal software*, or the software that you create to fulfill your customer needs, has no concept of enforcing a business model. Something must be done to it to prepare it for license management, and the way you do this can vary a lot for homegrown systems. Third-party license managers, on the other hand, generally employ two approaches to integrate the software with the license manager: injection or APIs.

- *Injection:* Given a "normal" application, the "injection" approach analyzes the object code and "injects" into it new code that typically obfuscates and/or encrypts the original code and adds the client license manager to enforce the license. In other words, injection works just like a virus, albeit a beneficial one. Table 4-1 lists some of the advantages and disadvantages of the injection approach.

TABLE 4-1 Advantages and Disadvantages of the Injection Approach

Advantages	*Disadvantages*
• Requires little or no work by developers • Can result in more secure protection, because the injection approaches obfuscate and/or encrypts code	• Increases size of code • Decreases execution performance • Can only be used with binaries; typically not suitable for interpreted languages

TABLE 4-2 Advantages and Disadvantages of API-based Approaches

Advantages	*Disadvantages*
• Provides maximum flexibility—you can control exactly how the license model works • Can be used with interpreted languages	• If not used properly can be easy to defeat • Creates lock-in to an existing vendor • Usually takes longer to implement a complete solution • Can lead to a false sense of security

- *API:* In this approach developers write to an API (or SDK) provided by the license manager vendor. The API provides for such things as license registration and enforcement (e.g., it has calls for "checking the validity of the license" or "logging in a concurrent user.") This approach is not strictly required for concurrent licensing, but it makes implementing such schemes vastly easier. APIs can be used with interpreted languages, but most vendors feel that using them in this manner does not provide very strong security. More plainly, it is relatively easy to hack the API approach in Java/C#. Table 4-2 captures the advantages and disadvantages of API-based approaches.

While professional third-party license managers have many virtues, you need to evaluate them very carefully. Consider the issues in the following sections when conducting this evaluation.

Business Model Support

To the best of my knowledge, no licensing manager supports all of the business models listed earlier. For example, I don't know of any that provide direct support for most hardware-based business models (such as per CPU or per-expansion-card). Most work best if you alter your business model to work well with their specific technologies.

Traditional license managers provide a fixed set of models with fill-in parameters. An example is a time-based usage scenario, in which you simply fill in the amount of time. More modern license managers provide a license scripting language, similar to languages like Visual Basic, that allows you to create customized scripts for creative licensing models.

Platform and Operating System Support

Make certain your license manager vendors can provide support for all required platforms and operating systems. When examining their supported platforms, take the time to explore their development roadmap, because, when a new version of an operating system is released, your development efforts are stalled until your license management vendor supports it!

Check Cracker Web Sites to Determine Solution Strength

It is easy to create a simple license manager. It is *very hard* to create an industrial-strength license manager that will consistently thwart crackers, maintain the security of your software, and ensure that you're getting the maximum revenue from your licensing model. If you don't have the skill to assess the strength of the solution, hire a consultant who does.

Check Backend Integration and Volume Capabilities

As stated earlier, the license generator is almost always integrated with backend systems. Examine your potential license manager vendor's ability to integrate its system within *your* environment. While you're doing this, make certain they can also meet your performance, volume, scalability, and stability requirements.

Make Certain Operational Environment Matches Yours

License managers create a whole host of operational issues. For example, customer service representatives may have to regenerate a license, or create a temporary evaluation license on the fly, or cancel a previously generated license. You'll have to make certain that the integrations created between your license generator and other backend components are sufficiently scalable and reliable. Make certain that your license manager vendor can meet your operational requirements. In other words, if your customer service environment is completely Web-based, make certain your license manager vendor can provide all required functionality via a browser.

Check Branding and User Interface Control

When the code enforcing the license detects a violation, chances are good that some kind of error message will be displayed to the user. Assess the degree of control you have over the content and presentation of this error message. You want to make certain that it meets all of your usability requirements, especially internationalization. You don't want to discover that your vendor has twelve dialogs it might bring up in obscure circumstances, and that you can't change any of their contents.

Examine License Content and Format

The format and content of the license should be understandable and should match your business requirements. Anything generated by your system, even if it is not used by the license manager, should be digitally signed. For example, you may wish to put

a serial number inside the license to integrate the license generator with other backend systems. Any custom or proprietary data that you store in the license should be signed.

Examine License Distribution Capabilities

Licenses can be distributed in a variety of ways. Make certain the vendor supports the approaches that are most important to you. Internet, e-mail, phone, and fax are the most common license distribution options.

Market Maturity Influences on the Business Model

The maturity of your target market is one of the strongest influences on the selection and management of a given business model. In the early phases of a given market, business models should be chosen so that they can be quickly and easily understood, primarily because you may not be certain of the best way to structure them. You may find that your customers prefer an annual license to a subscription, or that they expect discounts if they purchase in bulk. Moreover, despite the best intentions of the business plan, innovators and early adopters may expect and/or demand special terms.

As the market matures, chances are good that your business model will become increasingly complex in order to serve the idiosyncratic needs of different market segments. I helped one client whose growth had stalled attack a new market segment with the same underlying system simply by defining a new business model. The original one consisted of an annual license. The new one was pay per use. The easy part was modifying the underlying architecture so that both models could be supported. The hard part was creating the appropriate price points so that a given customer could choose the best model without harming the relationships with current customers.

The enforcement of business models also matches the maturity of the target market. In early market stages, enforcement tends to be lax. As the market matures, or in cases where you suspect piracy, the enforcement tightens up. My experience is that marketects and tarchitects take enforcement *far too lightly.* You've worked hard to create your system, and software piracy is a serious problem. Create a business model that identifies the real value provided to your customers, price it competitively, and enforce it accordingly. Just remember that onerous enforcement will lead to dissatisfaction among honest customers, so be careful.

Choosing a Business Model

Choosing a business model is one of the most challenging tasks faced by the marketect, as it incorporates everything that has been discussed in this chapter *and* several factors that are beyond the chapter scope, such as the business and licensing models offered by competitors (which may constrain you to existing market expectations) and corporate and/or environmental factors beyond your control (such as when another division does poorly and you need to find a way to increase short-term revenue). To

Hohmann_ch04.fm Page 94 Thursday, November 6, 2003 3:29 PM

help you through the potential morass of choosing a business model, consider these questions.

- *What is the target market? What does it value?* A crisp description of the target market and what it values is the first step in creating an appropriate business licensing model. If you're not certain of what it values, consider how it wants to use what you're offering. Once you've determined what your market values, show how your solution provides it.
- *What are your objectives relative to this target market?* In an emerging market you may wish to capture market share, so create simpler models. In a mature market you may wish to protect market share, so create more complex models to provide flexibility.
- *What is your business model?* Pick one of the business models defined above and customize it to meet your needs.
- *What rights do you wish to convey?* Begin by asking your legal department for a "standard" contract, as it will contain a variety of nonnegotiable rights and restrictions. See what you can do about everything that is left.
- *What is the effect of this business model on your software architecture?* Work with the tarchitect to make certain that any business model you propose is appropriately supported.
- *What is the pricing model?* The business model provides the framework for defining how you're going to make money. The pricing model sets the amount the customer will pay. You'll need to consider such things as volume discounts, sales and/or channel incentives, and so forth. Pricing choices may also affect your software architecture, so make them carefully.

As you develop the answers to these questions, you're likely to find that the best way to reach a given target market will require a variety of changes to your current business model, licensing model, and software architecture. You'll have to rank-order the changes in all areas of your product so that you can reach the largest target market. The benefits will be worth it, as creating the right business and licensing model forms the foundation of a winning solution for both you and your customers.

❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑ ❑

Chapter Summary

- Your business model is how you make money.
- Business models are associated with, and to a large extent define, license models.

- Your license model is the terms and conditions you associate with the use of your software.
- The most common software-related business models make money by
  - Providing unfettered *access* to or *use* of the application for a defined period of time
  - Charging a percentage of the *revenue obtained* or *costs saved* from using the application
  - Charging for a *transaction*, that is, a defined and measurable unit of work
  - *Metering* access to or use of the application, or something the application processes
  - Charging for the *hardware* the application runs on, not the application itself
  - Providing one or more *services* that are intimately related to application operation and/or use
- Business models associated with users (such as concurrent user licensing) motivate integration with corporate systems that manage users (such as LDAP servers).
- Make certain you understand every right associated with your business model. Separating rights may provide more opportunities to create revenue.
- License models may be enforced by home-grown or third-party professional license managers.

Check This

- ❑ Each member of the development team can define the business models currently in use or under serious consideration for the future.
- ❑ Our license agreements are congruent with our business model.
- ❑ Our license agreements define the specific set of rights provided to customers.
- ❑ We have chosen an appropriate mechanism for enforcing our business model.
- ❑ The costs of changing tarchitecture to support alternative business models are understood and communicated to marketects.

Try This

1. What is your business model?
2. How well does your architecture support your business model? Why do you claim this?
3. Can you demonstrate support for a new kind of business model? For example, if your current system is sold on an annual license, can you easily add support for some kind of concurrent license, in such a way that you can open a new market for your software?

4. If you're using a license manager, have you allocated enough time in your project plan to properly integrate it into your application?
5. If you are using a license manager, have you examined their development roadmap to make certain it supports your own?
6. Are your target customers likely to be innovators, early majority, majority, late majority, or laggards? How does this characterization affect your business model?