1

What Characterizes Rich Internet Applications?

"Let me just say just one word to you, young man — plastics!"

— Older male character giving investment advice to Dustin Hoffman's character, Benjamin Braddock, in the 1964 film *The Graduate*

First things first. This is a book for software developers by software developers. As developers, we are inclined to jump to the technical detail immediately. When we learn a new language, for example, we probably want to understand concepts such as typing, declaration, variable assignment, containment, and control flow structures: iteration, test and branching, modularization, and so on. This chapter is going to be a little different, but no less valuable.

This chapter lays a foundation for why learning the RIA design approach is important. In that sense, this chapter is going to be more "philosophical" than any other chapters in this book. It has only a couple code examples, primarily explaining the software revolution under way, and helping you understand the nature of the revolution. The rest of the book discusses code and coding; here you discover what's different and how you can profit from it. The discussion tilts toward independent software writers, by explaining how the competitive landscape is changing; but developers within the enterprise will benefit from understanding how to make their company more competitive externally, by building more scalable and user-centric public user experiences; and more effective internally, by building better collaboration and coordination tools.

In that spirit, read on to find out what makes RIAs (Rich Internet Applications) something worth paying attention to.

Rich Internet Applications are "Plastic"

The reference to plastics from *The Graduate* at the beginning of the chapter suggests the most famous meaning of the word "plastic," a commodity material which is often used as a substitute for the real (usually more expensive) material. In the film, plastics were being touted to the character as a good investment for the future. When we, as developers, look back at the formative days of this period, the next incarnation of the Internet, we may think "plastic" as well — in the dictionary meaning of the word: "transforming; growing; changing; dynamic, developing."

Thus, a first characteristic of RIAs is *plasticity*. The new generation of Internet-platformed applications is extensible and able to be recomposed into many different "look and feel" models, sometimes with the swipe of a cascading style sheet (CSS) paintbrush. Web applications, and Web pages before them, have always been "skinnable," and a simple demonstration of applying a cascading style sheet (CSS) in an HTML page shows how plastic (dynamic) an ordinary Web page can become. If you are well versed in Web page plasticity, feel free to skip this section; otherwise, read on.

An Example of Plastic (Dynamic) Web Pages

First, a Web page with no formatting: boring and mundane. Figure 1-1 shows a Web page with no applied styles. The information is there, but the page is static and lifeless.

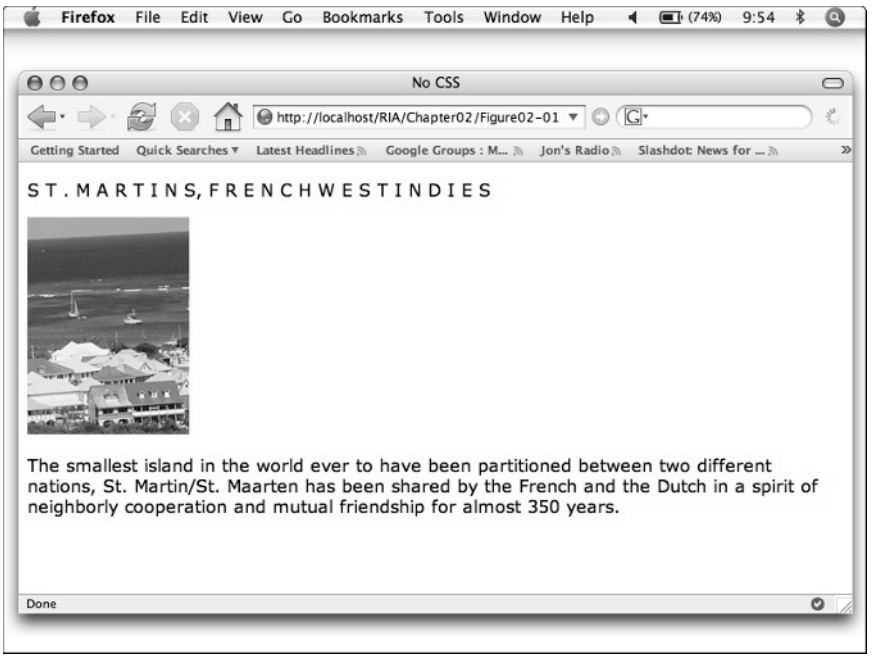

Figure 1-1

The following code creates the trivial Web page shown in Figure 1-1. The few DOM (document object model) elements are placed by default — a couple paragraphs identified by the and tag pairs and a JPEG image declared in the <img/> tag:

```
<html>
<head>
   <title>No CSS</title>
</head>
<body>
<p>
S T . M A R T I N S, F R E N C H W E S T I N D I E S
</p>
<img src="images/Scenic.jpg"/>
<p>
The smallest island in the world ever to have been partitioned between two
different nations,
St. Martin/St. Maarten has been shared by the French and the Dutch in a spirit of
neighborly
cooperation and mutual friendship for almost 350 years.
</p>
</body>
</html>
```

In contrast, applying a style sheet to the identical DOM dramatically changes the page to give it style and appeal, as Figure 1-2 shows.

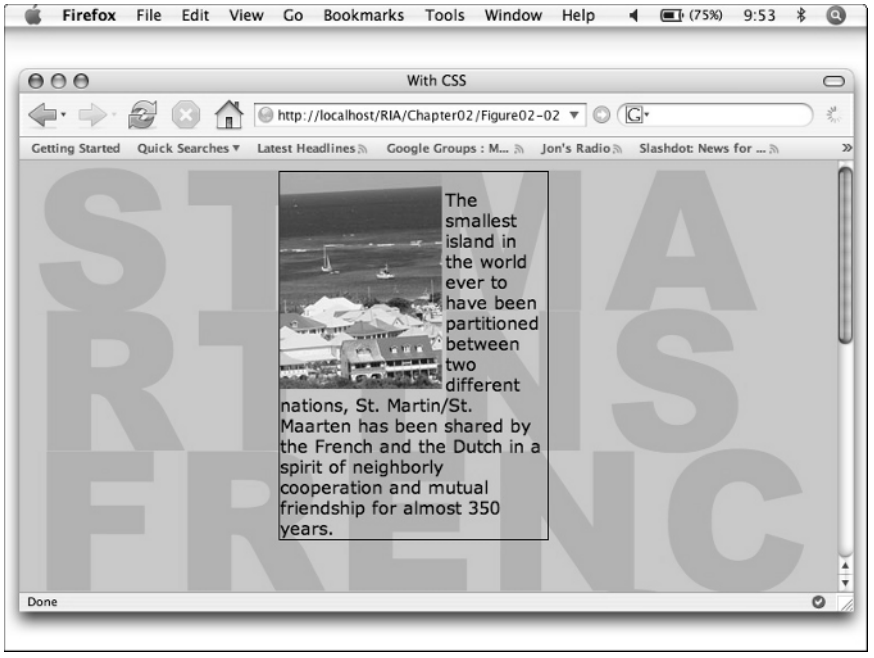

Figure 1-2

As the following HTML code shows, none of the DOM elements have changed, but a little style has been added to the page. The information content of the Web page is identical to that of Figure 2-1, but now the content stands out:

```
<html>
  <head>
   <title>With CSS</title>
   <style>
     #sxm {
      position: absolute;
       left: 10px;
      top: 1px;
       font-family:'Arial Black';
       font-size:180px;
       line-height:130px;
       color:rgb(173,181,184);
       word-spacing:-50px;
       margin:0;
       padding:2px;
       z-index:1;
     }
     #stm {
       position: absolute;
       left: 240px;
       border:1px solid #000;
       top: 10px;
       width: 248px;
       height: 340px;
       z-index:2;
     }
    </style>
  </head>
  <body style="background-color:rgb(194,203,207); margin:0; padding:0;">
   <div id="sxm">
    S T . M A R T I N S F R E N C H W E S T I N D I E S
   </div>
   <div id="stm">
    <img src="images/Scenic.jpg" align="LEFT">
    <p>
    The smallest island in the world ever to have been partitioned between
    two different nations, St. Martin/St. Maarten has been shared by the
    French and the Dutch in a spirit of neighborly
    cooperation and mutual friendship for almost 350 years.
    </p>
   </div>
  </body>
</html>
```

How Style Sheets Create a Better Page

You find the first secret to the improved page between the `<style> ... </style>` tag pair, where two styles, named `#sxm` and `#stm`, have been created. Additionally, now the informational elements of the

page are set into their own divisions, demarked by `<div> ... </div>` tag pairs. In these two examples showing a traditional use of the Web page, specific divisions are used (among other things) to show the browser where to apply styles. This is a simple but effective demonstration of the inherent plasticity you can take advantage of when writing the new style of applications, which you'll find in later chapters.

Divisions become even more important in designing RIAs, where they delineate segments of the page that modern browsers can dynamically update without a total page refresh. Throughout the book, and especially in Chapter 10, which describes the capstone application, you'll see many examples where dynamic partial page refresh creates a critical bond between the user and the application, and avoids the total page refresh, which made older-style applications seem static and more like filling out a form than working with an interactive desktop application.

> *This section showed a trivial example of page plasticity in Web 1.0 pages. To get a flavor of just how far designers can take pages, look at CSSZenGarden's page at* `www.csszengarden.com`*, for some really fun examples of plasticity using only CSS overlay for the identical content.*
>
> *To run either of the two example pages in this section, one easy method is to create a folder somewhere on your PC, edit each of the examples into its own HTML file, and then (assuming you have Python on your system, which Mac OS/X and Linux users do by default) start a Web server in the same folder with the following:*
>
> ```
> python -m SimpleHTTPServer
> ```
>
> *This starts a Web server capable of serving any pages in the same folder or child folders. If you don't have Python 2.4 or 2.5 on your system, you can download it from several sources, including* `www.activestate.com`*. Use*
>
> ```
> http://localhost:8000/<name of the HTML file>
> ```
>
> *as the URL for the page. For example, if the name of the page without a stylesheet is "noStyle.html", then point the browser at*
>
> ```
> http://localhost:8000/noStyle.html
> ```

We demonstrated plasticity in an old-style Web page to make the point that some level of dynamism has always been a possibility in the Web. Interactivity was another matter. Prior to the emergence of the Rich Internet, it was never that easy to provide interactivity, but Web 2.0 applications take basic elements of the traditional W3C specification of HTML and repurpose them to serve a far more interactive style of application construction.

> *A more dramatic, exciting, and compelling style of plasticity is accomplished by interfacing the APIs of various service-oriented applications to "mash up" your own application. Mashups take the idea of plasticity to a much higher level, as shown in the next chapter.*

RIAs also differ from the traditional Web in many important ways, and before looking at more code, let's talk more about these.

Rich Internet Applications: The Web Is Disruptive (Finally)

Often, disruptive technologies such as RIAs appear to emerge suddenly, almost at the speed of thought, with no obvious evolution. You could argue that in reality it's only notice by the popular press and a set of descriptive buzzwords that have sprung up in short order. The technologies behind Rich Internet Applications have been growing and maturing since even before the original Internet bubble.

Consider the enablers required for your new Web 2.0 applications to work at all. For one thing, in order for RIAs to exhibit the same level of responsiveness as their Rich Client Platform (RCP) ancestors, high-speed Internet access must be near ubiquitous. You can't offer a seamless user experience or convincingly separate the browser-resident user view and the server-resident model and controller logic without broadband, and when the Web was a toddler in the 1990s, such was not the case. Try using Google Maps over a 56 Kbps modem connection and you will understand the problem. Fortunately, you now have nearly ubiquitous broadband in the market, as well as customers who are your likely targets. Broadband is available almost everywhere in the U.S. At least 73 million Americans had high-speed Internet access as of April 2006.

Additionally, as mentioned earlier, the fundamentals of Web design have changed. Originally, design was about making pretty pages or replacements for magazines. Now design is slanted toward making things look like actual applications. Web design has become application design, and CSS have been repurposed to enhance the user experience, rather than the layout of the content.

Technology development takes longer than most of us think, with gradual changes that result in a perceivable difference taking years. The reasons for this are beyond the scope of this book, but gradual evolution, from the adoption of the lowly paper clip to the emergence of Rich Internet Application design and implementation techniques, can take decades. In truth, the underlying technologies that support RIAs have been around in one form or another for quite some time. In some ways, RIA development is a modern updating of the client-server model. The profound difference is the number of possible clients, given the explosion of wireless telephony-enabled devices, laptop portability, the proliferation of WiFi, and the next generation of PDA descendents.

From a very high altitude, you can view another RIA enabler, AJAX in the browser, as a simple trick by which round-trips to the server, as users input data that affects other fields/data on the page, implement "partial page refreshes." As mentioned earlier, modern browsers can refresh certain segments of the page (delineated by `<div>...</div>` tag pairs) without doing an entire page refresh. Partial refresh significantly improves the user experience by making application responses appear more seamless.

Neither of these technologies, whether in isolation or paired as they are in RIA, may seem like the basis for a revolution in application design and delivery, but consider some examples of how small changes conspire to promote dramatic change. E-mail and instant messaging (IM) are extremely similar in some respects. Both are essentially store-and-forward messaging strategies, after all. Both require a network to operate. In some cases, an e-mail and an IM sent at the same time to the same recipient can arrive at their destination nearly simultaneously, yet the end user perceives them as very different applications. Indeed, although they may share design characteristics, they have evolved to become very different applications with rather different monetization profiles and developer communities (if in fact e-mail can still be said to have much of a developer community). Looking at the difference between e-mail and IM is instructive if you consider the new generation of applications you can deliver as RIAs, and how they

are different from what came before them. Consider Table 1-1, which shows that two different software platforms may share similar design centers, but be very different due to the community that forms around them.

Table 1-1: Differences Between Instant Messaging and E-mail

Characteristic	E-mail	IM
Usage pattern	A formal enterprise communication mechanism	A convenient way to communicate whenever you need to say something to someone, usually in real time, and usually something immediate and possibly pithy
What it displaces	The business letter and formal corporate communications	The telephone, or even face-to-face communication
Ease of use	Requires a formal setup and system administration	Is always "on" with no effort on your part. You talk about "checking for e-mail," but you never say, "I am checking for IMs."
Immediacy	Slower and more ponderous; formal thought is required to compose a communication before you send it	Immediate, conversational, and social. You say what you need to say, when you need to say it. Although both platforms use a store and forward protocol, changing the speed of IM interchange creates a change in experience.
Application continuity	Episodic, with formal give and take and formal presentation or argumentation	Continuous, with the exchange of ideas just "happening"
Administrative burden to the user	You often feel the need to folderize and categorize e-mail for future reference.	You don't often review IM communication streams, although you could use search tools to review logged conversations. This distinction may be ending due to migrate-to-Web e-mail tools such as Gmail, or a RIA, where searching has replaced folder navigation.

You could argue that the differences between these two applications, which have similar technical underpinnings, are the differences between old school RCP or Web 1.0 applications and RIAs. RIAs offer invisibility, immediacy, a continuous streamed user experience, and "ad hoc-rity" (a lack of formal setup requirements). These are the disruptive qualities of Rich Internet Applications, each of which is explained in the rest of this chapter.

Rich Internet Applications Are Invisible

You are a software developer, yes, but at times you're also an end user, and as an end user of desktop applications, you are used to acquiring (sometimes even purchasing), installing, and maintaining applications, You look after their well-being, contributing a lot of unpaid labor to do the necessary "yak shaving"

(see the sidebar in this chapter) to get applications into the necessary state to "serve your needs." Figure 1-3 shows Google, a prime example of the invisible application. It's taken for granted, but it's never far from the end user's reach.

Figure 1-3

Invisibility of Google

Google is a prime example of the invisible application few can live without. Google's developers have deliberately made their software more valuable and more invisible by embedding its simple type-in box into the toolbar of modern browsers, and into the Windows System Tray for desktop search (Google Desktop), as shown in Figure 1-4. More than any other application, Google, has insinuated itself into the browsing experience so deeply that many users unconsciously invoke it almost as a reflex action. Unconscious use is the highest compliment users can pay to any product or service.

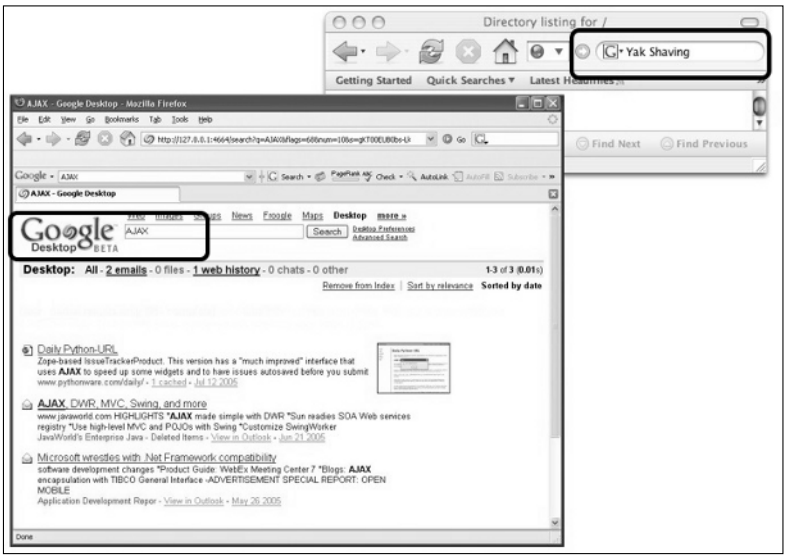

Figure 1-4

Traditional applications are a lot like a high-maintenance relationship in that they deliver value, but not necessarily in portion to the equity you invest. The more we experiment with this new paradigm, the more the kinds of applications we used to write begin to look clunky and primitive to us.

The more you use RIAs as an end user, the less you notice the fact that they are just "there," that they somehow disappear into the background. As a developer, you may remember the epiphany that occurred the first time you got facile with the many "applications" on the Linux (or if you're old enough, UNIX) command line. Although you rarely thought of them as such, `find`, `grep`, `ps`, `ls`, and `gcc` were indeed applications, and a shell "incantation" such as

```
$ ls -lat | grep html | wc -l
```

to count the number of possible HTML files in a folder was an early example of chaining applications in UNIX to create a mashup, or composite, application. After a while, though, you probably just forgot about them, used them to perform your work, and didn't even recognize them as software anymore.

Focusing Your RIAs

As you survey the universe of Web 2.0 applications, you'll possibly notice that individual applications don't (as yet) suffer from the bloat of native tools. They seem to focus on doing one thing and doing it solidly.

That's not to say it's impossible for you to write really bad RIAs, or that an escalation to bloated Web-based applications with needless baggage won't happen as a response to market pressure. It may. It's just that what has currently garnered the most mindshare and jumps out from the pack is notable for its simplicity. For an industry perspective on what it takes to fight bloat, see the sidebar "An Interview with 37Signals."

Two things may mitigate bloat and bad design, though:

- Bloated software won't be as quick to load or as responsive to use.
- Users will find switching costs much lower.

The advice often repeated in this book, and hopefully emphasized in the example code, is to design with a single feature in mind, and with the understanding that users do not want to "train up" on Web-based software. An important dictum to remember is that on the Web, user experience is paramount.

> **Eating Our Own Dog Food**
>
> One decision the authors made early on in the writing of this book was to use an AJAX word processor rather than a desktop tool, not just to "eat our own dog food" as we used to say during the dotcom era, but because we often needed to collaborate at a moment's notice, regardless of location, regardless of what operating system that was at hand. That way, we found that we didn't have to worry about having the "correct" software installed, or where precisely the content we needed to work with resided. Striving to make your software something that a large number of users take for granted only serves to increase your reputation and value. (See the section "How Am I Going to Make Money from RIAs?" later in the chapter for more information.)

Another reason why the transition between "applications" seems, from a user standpoint, so easy, natural, and seamless really has little to do with RIAs directly, but is worth mentioning: Tabbed browsers are most likely the desktop you always wanted but never got.

Users have settled for the chaotic Windows environment as though it were a fundamental law of nature. Developers went right along with this, "compounding the felony," as it were, but think for a moment: The windowed world of the traditional desktop is chaotic and switching from one application's space to another entails some pain because you're constantly searching, dragging windows around, and manipulating the file systems to get content into the right place for the benefit of the application.

Rich Internet Applications Break Down Walled Gardens

Platforms currently in use tend to control and channel intellectual property into a "walled garden." When you create a document with an RCP document creator, although you are the author, you don't own the content independently of the tool you use to create it. You can't write or read Microsoft Word documents without something that works remarkably like Microsoft Word. In fact, Word's vendor would rather you didn't use *anything* other than Word to interact with what should be "your" information. To take this logic even a step further, they really *insist* you use the operating system(s) that they will happily sell you to run the application that they also sell you, all to get at the information that really should be your information.

In the ideal world (from the *traditional desktop era software vendor's* standpoint):

- Content cannot exist without the mediating application. Vendors sell applications and store the content model in a format that prevents anyone from easily usurping their dominance over the user. The vendor decides when, if, and how bugs are addressed. Vendors need to create massive distribution systems that rely on end-user sweat equity to keep applications as up-to-date as the vendor decides they ought to be.
- Applications cannot exist independently of the specific operating system. The vendor will publish as much or as little of the system call stack as they deem sufficient to encourage independent software developers to develop the kinds of applications they want to support on their operating system. Generally, they will "bless" a certain few languages and dissuade others.
- Both the operating system and the application are tightly bound so that, at the vendor's discretion, both can be rendered obsolete. Vendors will do this ostensibly in the name of operating efficiency, but it also means that they can (and do) feel free to break the Model-View-Controller paradigm or inject other undocumented efficiencies into "house" code. It also means that the inner mechanism of the OS or application becomes arcane, brittle, and opaque to independent software vendors and developers (ISVs) outside of the organization. External developers thus struggle to come up with competing benchmark applications in addition to those they internally produce. Dominant vendors thereby protect their hegemony over the "cash cow" applications responsible for their revenue stream.
- Each tool creates an "island of automation," with its own data model, its own controller, and a view dependent on the OS and tool as platform. Because vendors are stuck in our worldview, they will have a hard time developing (for example) applications that are net-centric, collaboration-rich, OS neutral, or extensible — either by the end user or by other software developers.

- In marketing, developer conferences, and other interactions, vendors tend to convince both ISVs and end users that the trades-offs for these annoyances will result in assurance of (generally) seamless and smooth interaction (for the users), and protection and a ready market (for the ISV). They thus ensure a barrier to entry that keeps software prices artificially high and difficult to replicate.

It would be surprising if anything on this list is a revelation to you, either as a software developer or in your other role as an end user. These are the basic tenets of software, as it has existed for over three decades. In writing this book and asking you to look at new techniques, ask yourself whether the state of the world must be this way for some fundamental unalterable reason, or whether it simply has become a staid custom that limits our innovation and spirit of adventure.

In the traditional software world — the one that should be left behind — tools and applications have become traps for both the creator and potential consumers because they imprison content and intellectual property. A few words of caution, though: Just because you learn to write extensible, re-mixable, software-as-service applications in this book doesn't make you immune to the imposition of other forces just as inimical to open software as the old school OS and software vendors were. See the sections "Rich Internet Applications Are Network-Centric" and "Rich Internet Applications Are Browser-Centric" later in the chapter for more information.

Ironically, although consumer put up with walled gardens, software developers don't much care for them. Consider the wide variety of ways in which you can create and manipulate your Java/Python/Ruby code. VI, Eclipse, Netbeans, EMACS, and note padding applications (which exist seemingly beyond number) will do just fine. When you need a code viewer, open a terminal window, cat mycode.py, and you're done. One reason why you can operate in such an unfettered atmosphere is that ASCII is ASCII all over the world, and no single vendor can imprison the format. As long as you can create ASCII, you can program.

Rich Internet Applications Create New (Unwalled?) Gardens

The aim of this book is to motivate developers to write new generations of applications that create a different kind of garden. In this garden, the richness of the application is set free in order for content creators and consumers to interact with far less interference and human investment in acquiring and maintaining tools than the previous generations of "walled garden" applications. If the first incarnation of the Internet was about serving and presenting hyperlinked information, and the second was about submitting personal information to complete transactions, then the third is surely about creating, consuming, and manipulating content *in a social or collaborative setting*. Specifically, a generation of applications have emerged that are "us" powered.

The desktop generation of applications enabled end users to write, draw, print, and circulate their creations to a relatively limited circle of consumers, but today's content is potentially open to unlimited numbers of consumers, raters, taggers, and bloggers, who have raised the rate of content creation, consumption, and the potential for debate and discourse to new highs. Whole businesses are being created over the ranking of content by the "wisdom of crowds." Consider, for example, how difficult it would have been to create del.icio.us, Slashdot.org or dig.com in the desktop generation.

Constraints in the Unwalled Garden

The community of developers working with this new architecture (and you either are or soon will be a member of that community) has converged on a couple of critical concepts about how to create value with RIAs. Whether you build an application to create, store, and share content (too many to mention, but think Flickr), comment on content (e.g., Slashdot), collaborate on content (think Writely and this book's capstone application, "ideaStax"), or to remix content (again, too many to mention, but consider Google Map mashups, Apple or Yahoo! desktop widgets), the architect/developer chain of reasoning forces some interesting design constraints on new-style applications:

- Shareable content and dynamic content publishing are a foundational element of this new Internet age.
- You have less impedance to content creation and publishing than in past. To some extent, blogs replace magazines and books. At the very least, they expand publishing options.
- More shareable is better than less shareable and may even be monetizable.
- High-speed networks are pretty ubiquitous in the user community that consumes new-style applications.
- Given the ability to create a large community around specific content categories, there is a need for applications to reach large numbers of end users.
- An application has to resemble a radio or television network in its dynamics — except, of course, that those sorts of networks are "consume only." They don't permit the kinds of rich interactions that this new kind of network encourages.
- The end user will largely use a browser to reach an application.
- There is no monopoly on browsers. Browsers are supported practically everywhere (desktops, mobile phones, and various handheld incarnations); therefore, a new application doesn't have to write a vendor-specific API. In fact, it wastes developer cycles to write to specific client-side APIs, and it may sub-optimize the value of a new application to both the developer and potential end users.
- There is an industry-wide data transport API (HTTP), and an industry-wide display standard (HTML) for the client side.
- The application creator can still hold real value on the server side in the data model and the data manipulation logic (which need not be exposed to the application's user).

No formal committee sat down somewhere in Silicon Valley and decided that these principles would be the foundation for RIAs. There is no list like the preceding one formally stated anywhere in the literature, but the logic does seem both sound and fairly bulletproof when you consider the following:

- This book comes out at a time when it makes sense for developers to both write and read it.
- The characteristics for most RIAs observed "in the wild."

Some of these notions just evolved as a result of the availability of enabling technologies. Since the first Internet bubble burst, software developers went back to the drawing board with a strong understanding of the architectural possibilities. Innovators bootstrapped the current Internet "boom" by creating value around ideas and applications that would not have been possible before. The existence of this new architecture for applications creates a large "surface area" for content that was difficult to achieve in the desktop era.

Technically, the seamless experience of using Internet-based tools is one pillar of the next generation of applications. The ubiquity of broadband is one reason you can write distributed applications and not worry (as much) about responsiveness, as you did with desktop generation applications. The availability of unencumbered browsers is a ready-made outer container for applications.

However, there's more to RIA than simply moving traditional desktop productivity applications to a new venue. If you want to succeed at creating the next "must-have" application, you should keep the vision just laid out for you in mind: Expose the surface area of content-based applications to the largest number of "us" that you can. This, more than anything else, is your new design center.

At the time of writing, there were about 1 billion online content creators and consumers. These are the new end users; and if you make it easy for them, they will provide untold wealth in shared wisdom, social networks, personal perspectives, and a willingness to collaborate, all of which can be applied to your application. It's suddenly possible (maybe even for the first time in recorded history) to experience the massive cooperation referred to as the "gift economy" if you write applications that enable small contributions.

Understanding the Counter-Manifesto

The new kind of garden that you, the RIA developer, and by extension your end users, create can be stated in a kind of "counter-manifesto" that is used as the design center for the code example and the capstone application worked toward in this book. This counter-manifesto sounds remarkably different to the one described earlier for the desktop era developer. We present them as a set of application-level design patterns, which we enumerate in the next few sections.

Pattern 1: Expose (at least some of) the Remote API

RIA developers should create content in such a way that other mediating applications that other developers or end users might dream up can interact with it, display it, or repurpose its content.

> *You'll read more about this topic in Chapter 3, where you create mashups. Mashups and remixes occur when new applications (originally unintended) are constructed from data and services originally intended for some other purpose.*
>
> *For some interesting GoogleMaps mashups, check out* `www.housingmaps.com` *(houses to let) and* `www.mywikimap.com` *(cheap gas finder)*

Pattern 2: Use a Common Data Format

A different (but similar) pattern suggests that RIA developers create data models underlying the service-based applications so that they or another developer can leverage them. While the backend server may store the data model in whatever form that's convenient (whether that's a SQLite relational DB, an NFS mounted flat file system, or an XML data store is irrelevant), the output to the user view is expressed as something that any number of readers/parsers can handle.

> *It's cheating just a tiny bit to call this a "design pattern" and then suggest that you support it. Chapters 10 and 11 show that output to the browser really must be either XML or JavaScript Object Notation (JSON). Like the speed of light, this is not just a good idea; it's the law, duly enforced by all browsers.*

Pattern 3: The Browser Creates the User Experience

The only real limiting factor on application capabilities is the limitations of common commercially available browsers. You, the developer, can deal with this limitation because, as a trade-off, it's a lot better than the tall stack of entry limitations and barriers that drove you crazy during the previous software epoch.

If you need visual sophistication beyond the capability of the modern browser, you can use other browser pluggable alternatives, such as Flash. Additionally, in extreme situations, you can employ launchers for external application engines, although this may create a jarring discontinuity in the user's experience. One somewhat successful application-level pattern for doing this is Java Web Start, whereby the user navigates to a URL, which contains JavaScript for launching an application.

Pattern 4: Applications Are Always Services

Applications are part of a service-oriented architecture (SOA) in which applications are services, created from scratch or as artifacts from recombining other application facets. Subject to limitations (such as the availability of some other service on which your application may depend), SOA will create the software equivalent of evolutionary adaptation. Remarkably, no "protocol stack by committee" of desktop vendors was required to create this pattern. Further, unlike the UDDI/WDSL stack, the Web 2.0 SOA is simple and almost intuitive.

Pattern 5: Enable User Agnosticism

RIAs won't force the user to worry about operating systems or browsers. As long as the browser supports scripting in the browser, applications "just work." Developers could choose to take advantage of "secret handshakes" or other backend implementations that assure vendor or OS lockdown (and you can probably think of one browser and one OS vendor that does this), but the downside of making such a devil's bargain is that ultimately you, as a developer, will wind up expending more cycles when, after getting some user traction on a specific platform, you find that your best interests are served by extending you application to every other platform. It's better not to support vendors with their own backend implementation agenda, and to design from the start to avoid the slippery slope that leads to the trap of another walled garden. Thus, it's best to design to steer clear of OS and native library dependencies. If you do this, you can safely ignore much of the annoying Byzantine detail of writing applications, and specifically the parts that tend to make applications most brittle. Instead, you can concentrate on the logic of what it is that you're trying to accomplish for the user.

Pattern 6: The Network Is the Computer

Originally a Sun Microsystems marketing slogan from the 1980s, this bromide goes in and out of vogue. Java Applets exemplified it with the X11/Motif distributed model in the 1990s. Both those models ultimately failed, on infrastructural weaknesses. X11 failed because it exacted a huge performance cost on the expensive "workstation" CPUs of the era, and because its network model couldn't extend beyond the LAN with the broadband era still a decade or so away. Java Applets failed because of their size (long load time) and sluggish performance — again, broadband was a few years away.

This time, broadband is near ubiquitous in much of the cyber landscape. In addition, because the current RIA user experience is small and fits comfortably within the memory footprint of the modern browser, failure seems far less likely. Thus, the "operating system" for this new generation of applications is not a single specific OS at all, but rather the Internet as a whole. Accordingly, you can develop to the separation of concerns afforded by the Model-View-Controller paradigm, while ignoring (for the most part) the

network in the middle of the application. There still remain some real concerns about the network in the middle, and you do need to code to protect the user experience, given this reality. Many of these issues are covered in Part III.

Pattern 7: Web 2.0 Is a Cultural Phenomenon

Except for the earliest days of personal computing, when the almost exclusively male Homebrew Computer Club, which gave birth to the first Apple, defined the culture of hackerdom, the PC desktop has never been a cultural phenomenon. In contrast, Web 2.0 has never *not* been about the culture. End users rally around "affinity-content." Community is everywhere in the culture of Web 2.0. craigslist, Digg, and Slashdot exist as aggregation points for user commentary. Narrow-focus podcasts replace broadcast TV and radio. Blogging replaces other publishing formats and mass distribution formats. Remixing and mashups are enabled by the fact that data models, normally closed or proprietary, are exposed by service APIs, but they also wouldn't exist without end user passion and sweat equity. Some enterprising developers (e.g., Ning at ning.com) have even managed to monetize this culture, offering a framework for creative *hobbyists* who create social applications based on existing APIs. Chapter 3, which is about mashups and remixes, gives you a feel for what end users can do with the Web as platform, and modest programming skills.

When you create an application, you should consider the social culture that may arise to embrace your application. You can likely build in collaborative and social capabilities (e.g., co-editing, social tagging) pretty easily; if there's a downside to having an application exist only "in the cloud," the best upside is that a networked application also obeys the power law of networks: The application's value increases as a log function of the number of users. Most RIAs in the wild have many or all of these characteristics.

Consider the elder statesman of content-sharing applications, the blog, and its close relative, the wiki. Many blogs approach or exceed print journalism in their reader penetration, and wikis are far more flexible than their power-hungry desktop predecessors (such as Groove, for example) in supporting multi-user coordination and collaboration.

Rich Internet Applications Are Always Up-to-Date

A big reason for the lowered barriers to use (the "yak shaving" factor, as we call it) is that something rather dramatic, and at the same time something taken for granted, happened when end users began to seriously use invisible and pervasive applications — software upgrades, patches, security fixes, and massive distributions of service packs just stopped; suddenly they seemed an artifact of a bygone era. The reason for this major end user shift seems so simple now, somehow obviously "right."

As a developer, you can understand the profound and yet simple explanation in a way that end users perhaps can't: Because RIAs deliver a fresh instance of the application into the browser on reload, the end user always receives the latest version, with all fixes and updates, automatically, even if you just finished making the fixes last night. No massive distribution headaches; no complicated patches in the lethal grip of the end user.

Yak Shaving, and the Traditional Desktop versus RIA Development Styles

A huge plus for every end user situation you can contemplate involves lowering the barriers to use. A lot of traditional RCP application development as we have come to know it involves a great deal of what is now called *yak shaving*. According to Wikipedia, the term may have been coined at MIT's CSAIL. In any case, it refers to any seemingly pointless activity that is actually necessary to solve a problem that solves a problem, which, several levels of recursion later, solves the real problem you're working on.

A posting at `http://projects.csail.mit.edu/gsl/gsb-archive/gsb2000-02-11.html` gives another definition and an example:

. . . yak shaving is what you are doing when you're doing some stupid, fiddly little task that bears no obvious relationship to what you're supposed to be working on, but yet a chain of twelve causal relations links what you're doing to the original meta-task.

Here's an example:

"I was working on my thesis and realized I needed a reference. I'd seen a post on comp.arch recently that cited a paper, so I fired up gnus. While I was searching for the post, I came across another post whose MIME encoding screwed up my ancient version of gnus, so I stopped and downloaded the latest version of gnus.

"Unfortunately, the new version of gnus didn't work with emacs 18, so I downloaded and built emacs 20. Of course, then I had to install updated versions of a half-dozen other packages to keep other users from hurting me. When I finally tried to use the new gnus, it kept crapping out on my old configuration. And that's why I'm deep in the gnus info pages and my .emacs file — and yet it's all part of working on my thesis."

And that, my friends, is yak shaving.

You will probably find that developing RIAs involves a lot less yak shaving, or at least is replaced by a more satisfying activity — perhaps wool gathering about how you can make your application more "trick." Or maybe just sheep shearing, because 1) the technologies tend not to be as reliant on secret handshake system calls, 2) the objectives of most RIAs are more focused and less feature laden and therefore quicker to conceive and deliver, and 3) the code, test, observe, refactor cycle seems (to us anyway) to be shorter and iterate more quickly toward a finished product.

In addition, as a developer, when you tweak the server side of the application, you spend late nights fixing the data model or converting from one database to another. The end user is totally unaffected. Native desktop software never looks as appealing again and end user expectations are permanently altered.

Finally, as a developer, your expectations are altered as well. Because you don't have to concentrate on side issues such as distributing and maintaining multiple versions, you can concentrate on useful refactoring, or creating the follow-up project.

Rich Internet Applications Are OS Killers

Software has been tied to given operating systems since the beginning of computing. Originally, software was offered almost as a loss leader for a vendor to assure loyalty to mainframe hardware. In the age of the personal computer, users still could not buy just any personal computer to operate a given application, for even after some vendors mastered the art of selling software as a standalone product, application users were required to buy the supporting operating system and the approved PC architecture. If automobiles worked this way, an automotive application — say, getting from Washington to Baltimore — would require the purchase of a specific brand of car, perhaps with specific tires, and the car would run only on roads paved with asphalt, and this would be the only way to achieve the application's goal of making that trip.

Web 2.0 Layering

Thankfully, neither the transport system nor Web 2.0 applications have such rigid constraints. As shown in the graphic in Figure 1-5, RIA solutions are not quite so overconstrained. The natural layering of Web 2.0–networked applications has positive indications for developers — forcing good development discipline through separation of concerns; and positive indications for the growth and evolution of the network itself — it becomes much easier to insert new (as yet unanticipated) layers, and ultimately adds scalability, as pointed out in the "Interview with Industry Leaders" section at the end of the chapter.

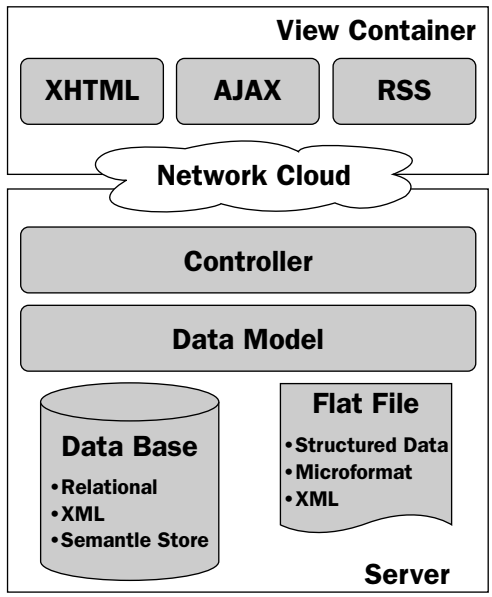

Figure 1-5

Thus, because RIAs use the Web itself as their platform, in the end it won't matter whether a user is running Windows, Linux, Mac OS/X, or something else. That's huge news for you as a developer. The design center of Web 2.0 requires the developer to deliver the application through a URL, via XHTML (the standards-based version of "good old HTML") into a browser. The Document Object Model (DOM) is used for dynamic display and interactivity without complete browser refresh. The developer embeds JavaScript inside the XHTML; that's the user experience side.

Cascading Style Sheets

The look and feel of the user experience is tailored through cascading style sheets (commonly CSS specifications in the XHTML.) The JavaScript logic in the browser uses a convenient feature built into all modern browsers as a way of making remote method calls (it's called an XMLHttpRequest, and is explained in detail in Chapter 3). The application's logic and the data model are often written in a popular framework (frameworks in Python, Ruby, and Java are covered), but a framework just buys you more productivity and less coding. There is no requirement to write to a framework — PHP, for example, is just fine. That's about all there is to it.

Leaving the Desktop Behind

In the desktop world, dealing with certain annoyances has become so ingrained in your life that you either hardly notice them or assume they are basic laws of the universe. Many of these seemingly immutable laws are part and parcel of the pain of working with applications in a native OS: compulsively hitting Save every few minutes as a safeguard against your word processor crashing; obsessively rearranging folder hierarchies; nervously running a virus checker a few times a day. RIAs embody a cultural shift for users, as well as a technological one.

In the event that users need to take smaller steps in leaving the desktop era behind, though, there is at least one effort to code an entire desktop in AJAX: AJAXOS (see `www.michaelrobertson.com/ajaxos/`. The differentiator for AJAXOS is that it assumes a network cloud, even for files on the computer platform on which AJAXOS runs. One might think of it as intentionally blurring the line between "local" and "in the cloud," in the belief that these options should always be a user choice. Everything, from the file system to the MP3 player, works this way in AJAXOS.

Is Java a Winner or Loser?

One question the authors of this book have been asked is whether Java is a winner or a loser in RIA, which is both a technological and cultural shift. We don't know the answer, but can hazard guesses; most developers would agree that delivering applications in Java on the browser side is a dead issue and has been for a long time now. As a comparison, run an RIA word processor such as Writely (docs.google.com) against a Java-in-the-browser applet version of Thinkfree (thinkfree.com). Although Thinkfree may, in the long run, offer similar features and responsiveness, in the end, the convenience of not having to deal with desktop issues wins the day for the RIA. It may be tragically ironic that Java, which in so many ways strives to be its own platform and manages nicely to be OS agnostic, and sparked the original move of applications from the desktop to the network, fails to be a significant technology of the future.

On the server side, it's another matter. Java is just as compelling as it is in the desktop application context. Additionally, Sun Microsystems, Java's inventor, may be one of the big winners in the cultural shift implied by the move to RIAs, simply because it's a server-class hardware vendor. For an additional perspective on Java versus agile languages on the server side, read the comments in the sidebar "An Interview with 37 Signals."

Finally, understand that AJAX brings it own brand of challenges: managing asynchronous operations and event-driven programming, especially if there's a need for multiple threads of control; network latency for these async requests; and coordination of multiple tasks on the Web interface, to cite but a few. You can read more about these and other challenges in Chapter 15. In summary though, RIAs are going to dampen future interest in both operating systems and "office suites."

Rich Internet Applications Are Browser Centric

The browser has evolved from an information-connecting appliance, to a modest application appliance (in Web 1.0), to a full blown application-hosting appliance in Web 2.0. And that's going to force us to think beyond the desktop. Whereas previously you only needed to test applications inside a software monoculture (for the most part), as a part of your development discipline, you now have to test with all major browsers, and at least the current and upcoming version of each.

Browsers are the "game changer" in the evolution of Web 2.0 applications, and you can highly leverage their native capabilities. As numerous code examples in succeeding chapters show, the simple ability to make remote procedure invocations using the built-in XMLHttpRequest from JavaScript has already changed so much of the "game." Consider how small a matter it would be for any modern browser to "up the ante" by incorporating work-alikes for the Microsoft Office, OpenOffice, Mac iWork suites. Already there are rich content editors — such as FCKEditor (`www.fckeditor.net`) and Tiny MCE (`tinymce.moxiecode.com`) — written completely in JavaScript and directly embeddable in the browser. In our capstone application, we will demonstrate an application similar to presentation composers.

Once content escapes the desktop and exists in the Internet "cloud," connected to the end user by wireless or broadband, the game has finally changed for good. TVs, phones, handheld devices, and devices yet to be conceived, are on an equal footing with the traditional PC or laptop. At this writing, most devices on the consumer market can host a browser. One operating system, Damn Small Linux, (`www.damnsmalllinux.org`), can boot from a USB thumb drive and host a full-featured browser.

Of course, you will still want to test every new application you write for browser compatibility. Some code for testing and working around browser peculiarities in various code snippets is available to you. The truth is that issues with browsers will likely decrease as AJAX development tools encapsulate the browser at a higher level of abstraction. Further, even mobile phone vendors (notably Nokia) are beginning to support AJAX-compliant browsers on their gear. Increased platform agnosticism will help bring your application to many more people globally than you would ever have attracted in the age of what has been called the software/hardware "monoculture."

Rich Internet Applications Are Network Centric

This is an additional reason why, as stated in the last section, RIAs work to deprecate the OS. Many end users are multi-locational and don't stop working just because they move from work to home. Nor do they set aside personal business when they move from home to work. There is no clear boundary between activities done in one location or another. As a result, end users spend huge amounts of time attempting to synchronize their home and work computers with both their content and the software that manipulates the content. The end value you create in the RIA space, whether it's the next great word processor or the next great personal information manager, enables your application user to access your software, and all their content, and is always available via a URL. Because of net centricity, sharing is easier as well. Consider all the end user tricks and accommodations that overcome distance and multiple computers: copying files to thumb drives, e-mailing content to Web mail accounts; using ponderous and

sluggish synchronizer applications (Groove, for example). All these activities still require intentional and conscious planning and forethought about just what content you need to copy for availability in multiple locations.

Rich client platform application advocates hold fast to the argument that the downside of RIAs is that if the network is broken for a particular user, then their application is broken too. It's certainly true, just as it is true that a virus-compromised computer loses its utility. At the end of the day, however, the network is far more robust and better maintained than the average end user's PC. Consider, too, all the net-centric applications we use and rely upon without giving them a second thought — satellite or FM radio, mobile telephony, and television. The end user base for Internet applications is already familiar with and comfortable with networked applications and devices. In addition, considering the number of U.S. cities that have already deployed or are contemplating municipal WiFi, there is a growing understanding that network ubiquity is a necessity of modern society.

An issue of apparent concern (for end users anyway) related to net centricity is the oft-heard comment, "I am concerned that all my documents are no longer on my own PC. They're up there in the Internet cloud. They're on the network, and that feels really dangerous to me."

Is this really true? In many situations, having one's "stuff" close at hand is a really bad idea: end users probably don't hoard their life savings under the mattress, or insist on squirreling away their chest X-rays in the hall closet. Nonetheless, PCs often give users a false sense of security and trust despite the fact that large numbers of laptops are stolen in a given year.

When end users raise this particular bogeyman, it's worthwhile to point out that tangibility is difficult to access. A user may well have content secure on my laptop, which (alas) has a bad power supply, bad disk drive, failed motherboard, or software virus. Their content is therefore tangible and even close at hand, but not accessible without the painful "yak shaving" required to resuscitate the PC.

One need only ask how many of them have lost valuable content to PC hard disks, or given complete access rights to indexers such as Google Desktop, or loaded "phone home" programs such as certain tax preparation software, or allowed a clandestine spybot such as the Sony/BMG rootkit to insinuate its way onto their PCs. Most end users would be far better off letting a high-security data center host their digital content, but the road to end user buy-in is long and only time will make a difference. The best advice here is to push end user education.

A final note on the topic of net-centricity and you, as a developer, is that you shouldn't assume that simply by moving applications to an RIA architecture that you are going to bulletproof your work and create a garden of delights, rather than a walled garden. In the U.S. in particular, end users may be trading one kind of walled garden for another.

As of this writing, much attention is being paid to the concept of a two-tiered Internet whereby ISPs would charge premiums for better quality of service and throughput. Certainly, the Internet is not, by its nature, immune to walled gardens. If an ISP can get its users (either by government regulation, by economic inducement, or maybe by mass hypnosis) to agree to let them mediate their browsing experience, then they can shape their users' perceptions, prop up political agendas, or sell users things of interest (to the ISP).

By way of example, consider that in 2006 the Chinese government hobbled searches and political discourse by tightly regulating ISPs. And the Digital Millennium Copyright Act in the U.S. exemplifies walled gardens that service either political or commercial agendas. Thus, the global development community's best intentions are undone by circumstances beyond its control. The best advice is to become engaged in affecting change in your country.

Political regimes are not the only ones seeking to create walled gardens around the users' experiences of the Internet. America Online was the earliest example of an ISP placing users in a walled garden. They remain one of the few that still does so.

Rich Internet Applications Are Mind-Altering

You can assert this statement in two different ways. First consider how your mind is altered as a software developer and a system architect, and then as an end user.

Observations from a Developer's Altered Mind

Most developers, especially those of us who have been cutting code for a while, have had the experience of mastering a new computer language and suddenly having our worldview significantly shift. When developers discovered C, they found that they could manipulate a computer's internal system state at a much less primitive level than before.

When Smalltalk came along, savvy programmers found that they could think in a message-oriented way—applications were "conversations" whereby messages were created and consumed. When Visual Basic was promoted to service the needs of the desktop era, it introduced the document as a visual container for the application, and many coders found it useful to think in document-oriented metaphors. When C++ and, more important, Java came into common use, an entire generation of software architects and developers suddenly started thinking in a more model-oriented way (one in which documents were but one of a host of models for architecting an application).

More recently, a small but significant number of developers started writing serious applications with dynamic languages such as Python and Ruby, and that introduced many independent and a few in-house corporate developers to agile programming practice. At every waypoint in the odyssey, software creators gained new perspectives that helped them design and write code differently, and write different code as well.

Now, suddenly, fueled by dynamic behaviors in the browser and agile languages on the server side, the wheel has turned again, giving developers yet another new perspective. The landscape that developers seem to see from this altered mindset is an impressive list of modern application design attributes. In this section, we present some observations on ways in which our perspectives have been altered. How many of these are exemplary of your Web 2.0 development or end user creative experiences?

Some Observations from Our Altered Minds

The first observation we offer to you is that the current perspective shift you may be experiencing is not due to the introduction of a new language, but of an entire platform. Python advanced a fresh philosophy of *creating general applications*. Those of you who know about the "Zen of Python" know this concept; and for those of you who don't, try invoking the Python interpreter and type in **import this** at the Python prompt to expose the interesting little "Easter Egg" shown in the left screen in Figure 1-6. In contrast to the "zen" of any specific language, we suggest on the right screen in the same figure that RIAs advance a new philosophy of *what applications are*; these observations are discussed in the following sections.

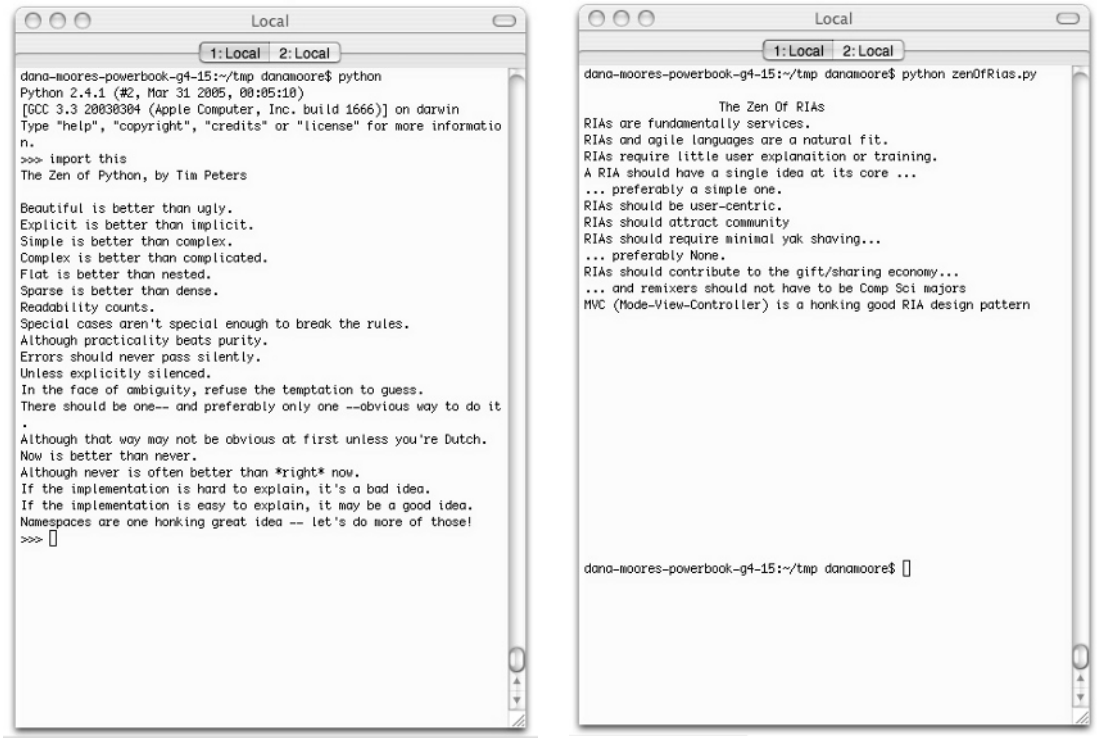

Figure 1-6

Rich Internet Applications Are Software as Service

Applications, according to our unofficial "Zen of RIAs" are first and foremost services. This means that they are invoked by URL navigation, rather than by installation, preparation, and configuration on a specific compute platform. If a RIA fails to just "be there," then you failed as a developer, and potential end users will toss it aside. Software as service in this case also implies open APIs, which in turn implies remixing and mashups. Mashups and remixes are made possible because many RIA developers have opened their APIs specifically to allow for remixing. In truth, they can hardly prevent it, as in general, server method invocations are exposed in the browser's JavaScript. Many vendors ask a developer to "apply" for an API key to pass along with a service invocation to authenticate the invoking application. You can think of this as a way for the original software's creator to keep score of the remixes and mashups using their API.

Rich Internet Applications Are User Centered

RIAs are user centered in two different ways, and you should design with each of these aspects in mind. First, they are user centered because they operate at a level above the operating system. This is important, because traditionally, a lot of technology products start with the internals and wrap the user experience around the technology. For example, most OSs have a lot of scaffolding that revolves around manipulating the file system, (there have only been a handful that don't, such as General Magic's Magic Cap). Most technologists start with an interesting architecture and build outward from that toward something that the user eventually sees, the user experience.

A really bad example of something like this might be a command-line interface for controlling MP3 playback. Typically, no one has put any thought into the interface, or because it was built from the technology "out," it's so rooted in the inner workings of the system that the only person who really knows how to use it is the person who built it.

In contrast, the only contact point the user has with a RIA is the user experience, so it's designed with this in mind. Notice, for example, how although Writely creates documents, there is no explicit need to save a document. Ideally, a user should just be able to move from one functional context to another in a completely modeless fashion.

> *For a perspective on starting with the user experience and moving inward toward the working mechanism, read the sidebar "An Interview with 37 Signals" later in the chapter. Modeless operation explains a little of Google's success in becoming indispensable to the end user. The Internet is far bigger than any individual's file system; hence, searching and tagging become the evolutionary replacements for folder navigation and folder management.*

Whereas Web 1.0 sites sought to look and feel like magazines, Web 2.0 sites all seem to have a different (but common) look and feel—big bold buttons and operable fixtures are the order of the day. That's because they are emphatically *not* books on the screen. Developers are not HTML publishers putting up their political rants or the next Great American Novel for people to read; they're putting up full-featured applications.

End users, for their part, seem to have become comfortable with composing their own swivel chair integration point. They inherently "grok" that browsers offer topical separation of concerns/interests that can be accessed or set aside at the click of a tab. The Web 2.0 "power user" doesn't boast Excel proficiency. Instead, a user controls three or four browser instances, each with multiple tabs, and boasts the ability to manage a score of RSS feeds.

Rich Internet Applications Are Inherently Collaborative

RIAs are also user centered in that communities of people participate in elaborating the creations of others. Whether it's commenting a blog, tagging a news story, adding metadata to a digital photo, or creating a local marketplace with craigslist, people are forming communities around Web applications, and many notable Web 2.0 applications are focused on community service in some way.

Although the authors' parent company, BBN Technologies, is a large-scale player in developing the formalisms of the semantic Web through our work in OWL, the semantic Web language, we have been terrifically impressed with "folksonomy," a simpler kind of tagging that seems to work wonderfully well, despite its simplicity and idiosyncratic nature. As Wikipedia explains it:

> *A "folksonomy" is a collaboratively generated, open-ended labeling system that enables Internet users to categorize content such as web pages, online photographs, and web links. The freely chosen labels–called tags–help to improve a search engine's effectiveness because content is categorized using a familiar, accessible, and shared vocabulary. The labeling process is called tagging.*

Collaboration via the Web creates a new kind of participation, a new kind of information experience in which a potentially world-spanning user community has the ability to modify their own online experience, and contribute to shaping that of others. In the capstone application developed in this book, we provide a framework for multiple people to collaborate to create content with content locking. Interestingly, this is still something that absolutely eludes even the best-known desktop applications (e.g., collaborative real-time word processing has never been done well outside one or two exceptional efforts, notably SubEthaEdit on the Macintosh; real time multi-participant coding hasn't either outside Netbeans Java IDE).

Rich Internet Applications: Small and Agile Development

The next generation of Web development is still an unfolding story as we write this, so it's difficult to support some of the contentions we would argue in this section; however, we can use studies of agile language development (e.g., projects developed in Ruby and Python) as a yardstick, first of all because the pre-eminent frameworks for Web 2.0 development implement applications in these languages.

Studies involving agile languages (see, for example, Lutz Prechelt's "An empirical comparison of seven programming languages." *IEEE Computer* 33(10):23–29, October 2000) show that developers write less code overall in agile languages and achieve results faster. The power of the individual developer or the small team seems to be amplified. In many cases that researchers have looked at, teams are smaller and more agile for whatever reason. It appears that using the frameworks we cover here, teams can be smaller and get things done faster: Writely, acquired by Google in 2006, consisted at the time of three engineers (the three co-founders). 37Signals, inventors and heavy users of the Ruby on Rails framework, consisted of four or five in 2006.

Rich Internet Applications Are Bound to Change

At this writing, new versions of all major browsers are in the works. The Web 2.0 revolution appears headed for a tipping point regarding numbers of applications converted from the desktop, in terms of both user base and number of deployment platforms. The shortcomings of the current generation of browsers and new feature requirements for the next will become apparent. This simply means that your knowledge and competence will have to evolve as well. Whatever changes are in store for browsers, they will certainly evolve to become even more general-application platforms than they are currently.

How Do I Change the Way I Develop Applications?

You may well be thinking that if, as we have implied, the application world around us is in the midst of changing dramatically, whether that implies you are going to have to change your current development practice to fit this new world. What are best current practices for the new generation of applications? Clearly, something is happening: Applications are "in beta" for longer than many formal versions of traditional software. Extreme programming and continuous integration, so recently avant garde, now seem possibly a little shopworn. In our survey of successful Web 2.0 efforts, there is a marked difference between traditional development styles, with formal release cycles and technology focus, and the new motif of continually evolving applications and customer focus. The following list describes some of the manifestations of this change:

- **All applications are betas —** It's not clear where this trend started, but as in so many other areas, Google may have led the way. From the end user standpoint, it really doesn't matter. If the utility of the application is sufficiently compelling, the end user really doesn't care about labels. Betas set positive user expectations and enable developers to take risks and sometimes even fail. Private betas and limited betas are part of an older style of cyclical develop and release events. Applications with cycles also imply an end to the support of a previous release of an application, and a not-so-subtle, implicit message to the end user that the developer desires to end the relationship with the user community.

- **Versions and bugs —** Just because "it's all betaware" doesn't mean you can somehow get away with shoddy implementation, however. The betaware strategy allows you to add features of an application when they're ready, and not tie new features to a largely artificial version rollout system. It's as important as ever to maintain good (CVS or SVN) versioning, building (Ant, Cruise Control) and bug-tracking (Bugzilla) systems, but considering what we just said above, you will have to set internal release milestones. Like a shark, you always need to be moving forward. It's not important what bugs were reported by users last week or last month; if a bug can't be reproduced against the live code, then you can safely kick it to the curb. One problem you can't afford to ignore, and which you have to design and plan for, is that while you *can* roll back code, you *can't* roll back user-contributed content. Finally, whether you provide the back-end server or use one of myriad servers for hire, you can't afford to let server-side bugs take down your site, so think about this nonfunctional aspect as much as you do about the functional capabilities of your application. You want users to think of your site and your application(s) like a utility — always on, always operating, just like a light switch. (Remember the invisibility thing?)

- **Test-driven development —** Test drive development becomes especially important because of distributed development teams, miniscule staffs and limited QA budgets, Unit, and Subsystem tests.

- **A RIA is a living lab —** The numbers of unique users that RIA sites attract often shock the novice developer or entrepreneur. RIAs can grow from no users to hundreds of thousands exponentially. Even a small sampling of users can be a large actual number. Your application is your creation. You should feel free to change features and arrangements of controls and find out directly how well users react. Building in click counters to understand how users navigate your application, which features they use, which features they understand, and how they navigate the application can all be done at a fraction of the cost of traditional usability testing.

- **Don't write one application, write two —** As explained in the next section, there are many paths to monetization, but understanding the community that accumulates around your application is one of the surest ways. Certainly, companies have attempted to categorize the people buying their applications in an effort to ensure repeat business or new product lines. The effectiveness of their intrusive market research styles — such as user surveys packaged with product registration — is debatable. Even in Web 1.0 applications, traffic monitors and other ways of "slicing and dicing" the user populations are generally ineffectual. In contrast, consider the Web 2.0 "connected culture." Users themselves identify their social networks and invite new users to share and otherwise become a part of the application's world. Flickr and YouTube are examples of applications in which the community and the application become indistinguishable. The kinds of user metadata that are useful to you in building value around your applications will certainly vary according to application and end user community, and you need to apply as much ingenuity to this aspect as the application itself.

How Am I Going to Make Money from RIAs?

If you're an enterprise applications developer, perhaps this is not a particularly relevant issue for you. For others, however, the question hinges on more than idle curiosity. At first glance, one might think that there is considerable exposure of intellectual property for your crafty rivals to inspect and emulate. After all, the user experience of a RIA (the "view" portion of the application), including the HTML, inline CSSs, and inline JavaScript, is available in clear text in the browser via any browser's "View Source" capability.

Certainly, it is very hard to violate the basic separation of an application's basic wiring diagram (the Model-View-Controller pattern) with RIAs. Remember, though, that the real intellectual content of an Internet application is secure within the server. How much of your service interface you choose to expose is up to you. As we will show in the next chapter, several "big name" providers expose a remarkable amount of their service call interfaces through external APIs. We will show you how to leverage exposed APIs to create applications that are mashups composed from parts of other applications.

As far as your own viability is concerned, consider that economic models are changing rapidly. First, the costs of production are really low. We have made a point in this book of demonstrating only frameworks that are available and support open source. Second, server costs are really low. In the United States, ISP costs continue to fall relentlessly. Further, when you begin to understand that with RIAs your application becomes a service, significant (nonfunctional) parts of your offering — storage, security, and always-on availability — are no or low-cost value adds.

According to Michael Robertson, the creative force behind ajaxWrite (one of the RIA word processors), millions of documents have been served to tens of thousands of end users. It seems clear that as more content manipulation capability moves "into the cloud," the days of the $400 "Office" suite are numbered. Further, it also seems clear that as a result of the "gift" or sharing economy that seems to be an emergent feature of the Web 2.0 culture, much of the content of new applications comes from end users. The value of Flickr or YouTube is not so much that end users can put their pictures or videos on the Internet cloud, but that they can publish their content to share with potentially millions of consumers. What the developers behind Flickr and Writely have done is simply to provide a framework to load, store, and tag content.

Although all sorts of models exist for creating value in return for compensation, the observation to readers is this: Don't use Microsoft as a reference point. The timeworn models from the desktop era may well be on their way to the dustheap of history. Further, build different things — things that play to the inherent advantage offered by the fact that you are building a networked application, one that provides a framework for multi-user contribution and collaboration. Remember that you may well want your application to be "invisible," enabling user content creation but building an air of indispensability around your application.

How much value is there to becoming so indispensable that you are taken for granted? As pointed out earlier in the chapter, the one application most end users take for granted, and consider to be a part of the essential fabric of the Internet, is Google search. Google's market cap in April 2006 was $128 billion USD.

Google's approach is that while they help end users locate content of interest, they also help advertisers locate "eyeballs" of interest, by assuming that an end user's search *might* be a search for goods and services. Most often it isn't, but Google doesn't need to be right much of the time, even most of the time. In 2005, advertisers sent $9 billion USD with the U.S. broadcast networks, where almost all their message was irrelevant to almost everyone. Google's AdSense capability can at least make a somewhat educated estimation of consumer interest. Perhaps even the advertising community is catching on — during 2005, Internet advertising revenues increased 28%.

The message here is that powerful things can happen if you build a target community around an application. Consider, for example, the potential for an application built as a volleyball-centric website — supporting league scheduling, result reporting, team statistics, and so on. Such a site, even with a relatively small user community, would be extremely valuable to equipment producers. Other Internet communities (e.g., del.icio.us, ClipMarks) create data sources that appear only on the Internet; the value of those data sources is currently underexploited, but you should recognize that there is value.

Rich Internet Applications Are Seductive

For end users, RIAs are certainly seductive, given the mashup capability for so many applications. Further, as suggested previously, they are seductively easy to use. Here, though, what we really mean is that they are seductive to us as developers. It seems much easier to get from thought to working application. Online tutorials for TurboGears and Ruby on Rails show a developer how to get a better than Hello, World quality Web 2.0 application coded and running in pretty short order, and that can be enough to show a novice developer the overall structure of these really fine frameworks. In truth, there's much more involved in doing nontrivial works, but the initial "seduction" provides a real "developer rush."

> *Ruby on Rails' captivating screencast boasts of "Creating a weblog in 15 minutes" (www.rubyonrails .org/screencasts); TurboGears' equivalent is "The 20-Minute Wiki" (www.turbogears.org/docs/wiki20)*

Because RIA development is most often associated with agile languages whereby often a small amount of code pays bigger dividends, milestones seem to happen faster. Those of us in the agile languages camp think we know why this is so. Agile language developers have always been a little suspicious of the *BIG DESIGN* waterfall method. That's the technical approach that always seems to produce a lot more UML than code. It always seems to such developers that having a well-defined application description in mind, plus the time-honored extreme programming approach of "design a little aspect, code a little, test a little, refactor, and repeat," yields better and faster results than endless agonizing over decomposing the big system picture.

In our experience, Web 2.0 design seems to strike the right balance between doing some up-front planning and modifying applications as experience moderates both the goals and the approach. Further, the frameworks that we cover in this book (Ruby on Rails, TurboGears, Django) make it eminently possible to do just that — incrementally design and deliver. That's not to say that *some* planning is a bad thing. The practices developers use commonly — separation of concerns, decomposition of the problem into object, and so on — still apply, and we will follow those precepts throughout the book (and, we hope, succeed).

Even when used for creating OS-bound applications, RIA standards may be easier to write for than desktop APIs. Slowly, the Macintosh Dashboard and Yahoo Widgets, the great-grandchildren of the old terminate-and-stay resident (TSR) applications (like the venerable DOS SideKick) are starting to gain some developer notice, and one reason is because, having mastered the skills and adopted the RIA mindset, it's hard to return to the bad old days of boring code. We cover desktop RIAs beginning in Chapter 17.

Interviews with Industry Leaders: Jason Fried and David Heinemeier Hansson of 37Signals

This chapter makes a number of assertions regarding the Web's future shape and potential beyond its first incarnation, and you may agree or disagree with at least a few of them. We drew many of our conclusions from listening carefully to industry leaders and revolutionary thinkers. From time to time throughout the book, we'll bring you insights from authoritative voices themselves. We interviewed Jason Fried and David Heinemeier Hansson of 37Signals, (37Signals.com), leaders in creating value in the Web's next generation.

On Being an Early Web 2.0 Innovator

Q: *You started out as a Web design company. Now you are delivering the Web as an application platform. When did you realize that things were changing on the Web from a hyperlink information browser to the application framework we are beginning to have today? Was there ever an "Aha!" moment or was this just a natural progression from Web design to application design?*

37Signals co-founder, Jason Fried: I don't think there was ever an "Aha!" moment, really. We needed to build a product for ourselves. Basecamp was built for ourselves originally. We decided that we needed to put it on the Web because we knew it needed to be centralized. Desktop software wasn't something we knew how to write, and desktop software really doesn't work well for centralized information. We felt that the Web was what we know best. It wasn't that we had a long-term vision, where we thought everything was going to the Web. It was just something we did out of necessity, and also we knew how to write Web-based software. We lucked into doing the easiest things for us and that happened to be doing Web stuff at the time.

Q: *Have the other projects emerged out of Basecamp or are those projects that you thought would be relevant to your own company and then you started to expose those as external services as well?*

Jason: Everything we built we built for ourselves first. We always filled a need that we had and we realized, as we like to say, "We are not unique snowflakes." If we need a tool, then so do hundreds of thousands of other companies just like us. There is nothing special about us that would only make a certain tool only relevant to us. So we say, let's put our own time into building this, and perhaps let's turn it into a product because other people need it as well and let's generate some revenue off of it.

On Managing to Make the Software Subscription Model

Q: *You seem to have hit a golden mean in which people don't give a second thought to using your software through subscription, and yet one of the world's largest software vendors has tried that and broken their pick on it for more than a decade. Why do you think you're succeeding?*

Jason: I think it's because our value proposition is significantly different . . . and that is fair pricing and good products — products that don't do a lot of [unneeded] "stuff"; products that just do a few things well and then get out of your way. That's what's compelling about what we are doing for people.

If you look at a lot of other software companies — they are in the "software business," so they think they need to build more and more software that does more and more things to justify their cost and justify the belief they are the greatest software company in the world. More features doesn't mean that they deliver what people need. They really need to focus on simpler things that enable people to work better. We don't worry about what everyone else is doing and we make the pricing fair. For example, the entire software industry prices everything by seat, or by user. If one person uses it, it's *X* dollars, but if 10 people use it it's 10 times *X* dollars. Instead, we say, "Look, have as many people use it for the same price.

We don't exact what we call "a participation tax," which is what just about every other software company does. When you start doing that you are discouraging more people from using your product. All those things coming together is giving us a nice head start on the software subscription model.

On Maintaining Focus

Q: *It seems like one of the common threads in interviews with 37Signals is to keep it simple, don't bother the user. Now that some of your products have been out on the Web for a while and you've gained more users, you've probably gotten a lot of feedback. Do you have to actively fight the urge to "feature creep" your applications.*

Jason: It's certainly a challenge and takes a lot of discipline to say no, but we have it in our heads to say no to everything, including our own ideas, first. That's not to say that people won't say yes later, but initially we say no by default to every new idea, and that allows us to be selective and keep listening and hearing more requests; and if we keep hearing that from a large number of users, we'll say yes ultimately. We don't react to one person's request, not even our own. When you have hundreds of thousands of users, you can't respond to a single request.

It definitely requires a good bit of discipline to keep things simple, and the software industry, in general, doesn't have much [self] discipline. As products mature they get more and more complex for most companies. I don't feel that is the right trajectory. We are learning all the time.

On the Role of Agile Languages in Rich Internet Applications

Q: *How did you guys come across Ruby? It would not have been the first choice on most peoples' lists a few years ago even if they were agile language savvy. Something that bothered people awhile back was that Ruby didn't have a native way to deliver user experience. Java has Swing. Python has TK, Wx, PythonCard, and a ton of other UIs. TCL has libraries, but Ruby had nothing but the Web. So how did you decide on Ruby?*

Ruby on Rails creator, and 37Signaller, David Heinemeier Hansson: Ruby is a language for writing beautiful code. Beautiful code is code that makes the developer happy, and when a developer is happy they are more productive. Ruby is unique in the way it lets you express something briefly, succinctly, and beautifully, whereas both PHP and Java are either verbose or ugly in their structure and the way they structure code.

On the Role of Frameworks in Rich Internet Applications

Q: *Have you seen the Java framework from Google (http://code.google.com/webtoolkit/)? Do you have an opinion about that? Have your looked at TurboGears (www.turbogears.org) or Django (www.djangoproject.com/)?*

David: Yes. First, the Google stuff is still Java, so it's basically uninteresting to me in its raw form. The approach they are using is something that we've been championing for about a year now, which is to write JavaScript in another programming language other than JavaScript itself. We have in Rails something called RJS. We write JavaScript in Ruby and that gets compiled at runtime to JavaScript. It's great to see Google follow through on that.

As far as TurboGears and Django go, both are simply interesting frameworks. They have not been out as long as Rails and definitely don't have the same momentum as Rails, and also they are written in Python, which is a great language (but isn't Ruby). On the scale of things, it's much closer to Java than PHP. I'd rather be writing in Python than Java or PHP.

Django seems more interesting. TurboGears is basically a "glue" stack which takes existing components and "glues" them together in a way that will fit a solution. A big part of the success of Rails is that it is not a glue stack. All the components are developed knowing each other, and that's the big pain we are

alleviating from the Java world where everything is a glue stack. You have to go out and hunt to get a ton of different frameworks and try and glue them together. I think that the Python frameworks are following the same approach we are, which is that if you use a consistently productive language, you can make everything yourself, and you'll be better for it.

On Agile Languages and Scalability

Q: *With your selection of Ruby do you have concerns about scalability at all?*

David: That's an easy answer, because it's exactly the same as other open-source languages. It follows what we'll call a *shared nothing architecture*. This means we split everything across three layers: Web, Application, and Database. Then make sure you don't keep state in your Web layer or in the Application layer. When you do that [adhere to these conventions — ed.] you get the wonderful benefit of being able to add unlimited Web servers, and unlimited application servers because all the state is being pushed down either to a DB, a shared memory layer, or NFS storage, so it's not your problem anymore.
That is basically the solution we picked. Scalability on data is hard. We don't want that problem. There are people out there that solved that problem long ago. Database folks are some of these people. So, scalability — it doesn't have any issues at all with respect to agile language. It's infinitely scalable at the Web and application layer — just add more servers.
The [Ruby] language itself is slightly slower than some other languages. It's in the same ballpark as Perl, PHP, and Python. Sure, you can hand-tune some Java code or write something in C or C++ that is faster. That's great, but it doesn't matter too much. We are in a time where CPU cycles are getting cheaper by the day, but programmer cycles are not getting cheaper and haven't gotten cheaper for a long, long time. The parameter you should be optimizing for is to get more programming cycles out of the development team. Using a more productive language and framework is the correct way to do that.
Just as a performance metric, we know that we are processing millions of requests a day. We're doing very well at that and plenty of other sites out there are supporting hundreds of thousands of users.

On Creating a New Rich Application Development Environment

Q: *What are the general guidelines you use to determine whether a new feature or concept makes it into Rails' core? Is it a benevolent dictatorship like Linux? Or a group decision?*

David: It's a group decision, but there's always a shadow of a benevolent dictator, me. If it's something I don't like, it's not going in. It's a simple as that. Most of the time, after using Rails for some time, they get the culture and they understand the motivation and what matures and tends to lead to patches in line with that vision.
Q: *Where do you see Rails going as it evolves?*

David: It continues to be a collection of solutions to problems encountered by the contributors. We don't have a road map. We don't intend to make a road map. We don't develop features for other people. The only features we develop are the ones we need ourselves. That is the only way to get an effective framework. If you try to imagine what some other programmer might need someday you are treading on thin ice.

On Working from the User Experience as a Functional Specification

Q: *We've heard it said that it's a 37Signals philosophy to start from the user experience and, using that as a functional spec, turn traditional application design on its head. Spending 80% of the time working on the UI seems counterintuitive. Was that a personal sensibility or did you arrive at that because you stated out as a Web design firm?*

David: Yes, that definitely helped. Once we started not trying to live up to other people's expectations of how you are supposed to do things, we got more things done and we got more right things done. That grew of practicing these things.

On Mashups

Q: *Something that amazes people new to Web 2.0 is the ease of doing mashups. You seem to be somewhat against exposing too much of your API so that other people can do mashups.*

David: Most of our products have exposed APIs, but there is a lot more "talk" than "walk." We haven't seen a large number of mashups that have been terribly useful rather than just cool. On the grand scale, exposed APIs don't matter as much yet as people would like us to believe. Our direction with Rails is to make it easier to do APIs right out of the box.