

New to Lithium

An Introduction to the PHP Framework

David Golding

Audience

This introduction is intended for beginners to PHP frameworks in general and to the Lithium framework in particular. The following types of readers will most likely benefit from this introduction:

- PHP developers who haven't used a framework before
- Novices to Lithium, CakePHP, CodeIgniter, Symfony, Zend, or even non-PHP frameworks like Rails or Django
- More advanced PHP developers looking for a quick introduction to Lithium but aren't expecting me to out-do the official documentation
- Readers looking to try out a new code paradigm like Model-View-Controller, Aspect Oriented Programming, or high-level modularity
- Readers that are scratching their heads after reading the previous bullet point

Here's a quick litmus test to determine if this introduction is too basic for your taste: if you can download and install Lithium, and create a basic web application with it, say (how clichéd is this?) a blog or social profiling system within an hour or two, you're probably too advanced for this modest introduction.

The following kinds of readers should proceed with the knowledge that I'm disclaiming anything noteworthy for them in the pages that follow:

- Bachelor of Science graduates with any kind of degree in computation, computer science, or information systems
- Working, professional developers
- Mathematicians
- People who churn out code better than their own spoken language
- (You get my point)

While I'm at it, here's another litmus test for you: if you regularly use a code repository and collaborate with other developers, chances are you have enough experience to feel like this introduction is a waste of your time.

So with that, dear reader (if you're still with us), let's take a look at this piece of PHP coolness called Lithium.

Say Goodbye to Sucky Code

One day I got a call from a trusted source asking for my help in migrating a website of his. Unfortunately, his pointman for this application backed out at the opportunity to really expand the site because the feature set was approaching a high degree of complexity. I had to hand it to this person—when faced with a lucrative account, he stayed honest and admitted that it was beginning to get over his head. This novice developer had never intended to build enterprise-level applications and had stuck to a business model favoring more simple website design and hosting.

Because of this scenario, I was called upon to migrate the site off its previous host to a new one. Easy enough. It had been built in PHP and used a MySQL database as its data source. I felt right at home. No problem. I assumed that I could do this sort of job in a matter of minutes, at worst maybe an hour or two.

I was terribly, horribly wrong.

What should have been a simple process of migrating a website from one remote hosting environment to another one *with exactly the same infrastructure* turned into a behemoth of badly written code. We've all been there when the system throws errors relating to unsupported extensions or missing dependencies. These can slow down a migration task to a standstill. The tragedy of this was that all of the problems I encountered during this migration had to do with a cascading flow of one inconsistent piece of code after another.

Without exaggerating things, literally the entire web application—and this was a very expansive piece of software that involved multiple processes across multiple employees and users—was a single series of business logic. Yes, there were separate files, but there was no harmony to them whatsoever. Require and include statements appeared ad hoc. Locating a mismatch in the code base was like combing a hundred yards of beach for a lost ring. Filenames held no semantic or relational value. It was impossible to discern from a look at a folder's contents which files mattered for what processes, and no rationale seemed to inform why a given library or class or script or resource went in a given location.

I get chills writing about this episode in my programming life. The bad ones, not the good ones you get when beholding [fireworks](#) or driving a golf ball 300 yards up to the green and a chance at eagle. You know what I'm talking about if you've ever had to work with sucky code (hopefully written by somebody else!).

Lithium unequivocally and unabashedly takes non-suckiness as its chief starting point and reason for being. If you actually like building entire applications in business logic alone or programming PHP like it's glorified HTML, then Lithium can do very little for you. You'll probably get frustrated with it the way I get frustrated when reading the theoretical mathematics of Kurt Gödel or the philosophy of Martin Heidegger. Truth is, my friend, if this is how you insist on doing programming, then Lithium really will be way over your head. It can haz more smart than you.

But no fear: here is your chance to come out from underneath your rock and embrace the joy of sticking to a well crafted system. Adhere to Lithium’s software design principles, and you’ll most certainly encounter the good chills that we programmers sometimes get when watching beautiful code transcend redundancy.

The Whole Paradigmatic Package

I should assume better of you. You’ve come to this in the first place because you know something about good programming, otherwise Lithium would be far off your radar. But for good measure, I wish to get a little abstract at the outset of our discussion. Rather than start with some installation tutorial, I’m going to get perhaps a little professorial. This is the inverse of how most technical writers approach the first stages of technology instruction—they love to have you hello-world before telling you what you’re getting yourself into or even why you are hello-worlding in this language or platform or framework as opposed to some other tool.

Lithium is as much a *paradigm* as it is a framework. Keeping the length of each line within 100 characters is as much a part of Lithium as creating a file called “Profiles” and having it extend the Model class used for all your database handling. Just because you can make the Controller class do something doesn’t mean you should—in fact, often it means you shouldn’t. Lithium comes along and incentivizes the separation of concerns. If you understand Lithium’s own paradigm for housing concerns, aspects, and modules, your applications will run more efficiently and you’ll find Lithium’s libraries fitting into your own code rather seamlessly. This is all to say that Lithium doesn’t stand by as a random collection of supportive libraries that you can include in your code where it helps. Lithium actually has the ability to integrate completely with your app, thanks to how it loads classes and provides key filtering methods.

That may be a bit much. We haven’t really even gotten to anything yet. However, it’s at the outset that I want to be clear about the course ahead. You are not going to get a lesson in building applications and making Lithium do things. This is thoroughly a lesson in *designing* aesthetic, robust, and scalable code architectures.

Let’s Do This Thing

Enough with the introduction to the introduction. Let’s fire up Lithium and see how it performs.

I’m assuming a couple of things. First, that you already have a web server running on a localhost that you can work with. You should already have PHP 5.3 (hopefully the most recent stable release of PHP 5.3—I’m using version 5.3.6). You should have a data source to work with. I’m going to stick with MySQL because of its ubiquity among beginners. I highly recommend, however, that you explore a NoSQL approach. Lithium is particularly well suited for MongoDB, so there may be a good number of you working in that environment. If

so, I'll assume you are already familiar with relational databases enough to translate the concepts across the differing data environments.

Go to Lithium's code repository on GitHub and check out the latest version. For a permanent link, visit the official website of the project, <http://lithify.me>. At the time of writing, the repository is located at <https://github.com/UnionOfRAD/framework.git>. You can run `git clone` to download the master branch to your own desired location. (For further details on this, see Scott Chacon's fabulous book, *Pro Git* or this chapter from the book at <http://progit.org/book/ch2-1.html>.) If Git isn't playing nice for you or just isn't your thang, just download the latest stable release from lithify.me, or use the download links on the GitHub site. Heck, if you're a Mac user, you can even run a simple desktop application to do all the Git processing for you (just click on the "Clone in Mac" button to get started).

You should download from the `framework` repository, not the `lithium` one because it contains the application folder structure as well as the Lithium libraries. You will need both to get running. After you have a running Lithium application, you can update the Lithium core using just the `lithium` repository, this way you won't override anything in your `app` folder.

Hello Lithium

It is expected that you will rename the `app` folder included in the parent `framework` directory and manage your server in such a way that the `app`, `libraries`, and `webroot` folders are distributed in a secure fashion. In other words, it's bad practice to have the application libraries or the Lithium core anywhere accessible via your webroot, so let's put these somewhere that suits us better than the current structure. Then we can configure Lithium to work with that more customized environment.

I like to house all of my server's libraries in one location. You will likely choose a different site for this, but I've placed the `framework/libraries/lithium` folder in my `/usr/lib` directory, and I've ditched the outer directories in the process. I've also renamed `app` to `tutorial_app` and have placed that one level outside of my server's webroot. I've taken the `webroot` folder out of the `app` folder, and moved this into the webroot with the new name `tutorial`. When I fire `http://localhost/tutorial`, it should call my new Tutorial application.

Now that I've done this, I have to run a couple of configurations to get it to play nicely. Depending on your own web server's configuration, you may need to do a few other action items to get this application to run. As that is in the realm of web services, I'll leave that to other books and manuals and question boards to explain.

First, open the newly named `[webroot]/tutorial` folder and locate the `index.php` file. It contains only two lines of code, and the rest are documentary comments. Around line 21 is the `require` statement that brings in the `bootstrap.php` file responsible for hooking together your app and the Lithium core. Since the folder structure has been changed, this line will break and throw a PHP require error.

Change this line to something like this:

```
21 require dirname(dirname(__DIR__)) . '/tutorial_app/config/bootstrap.php' ;
```

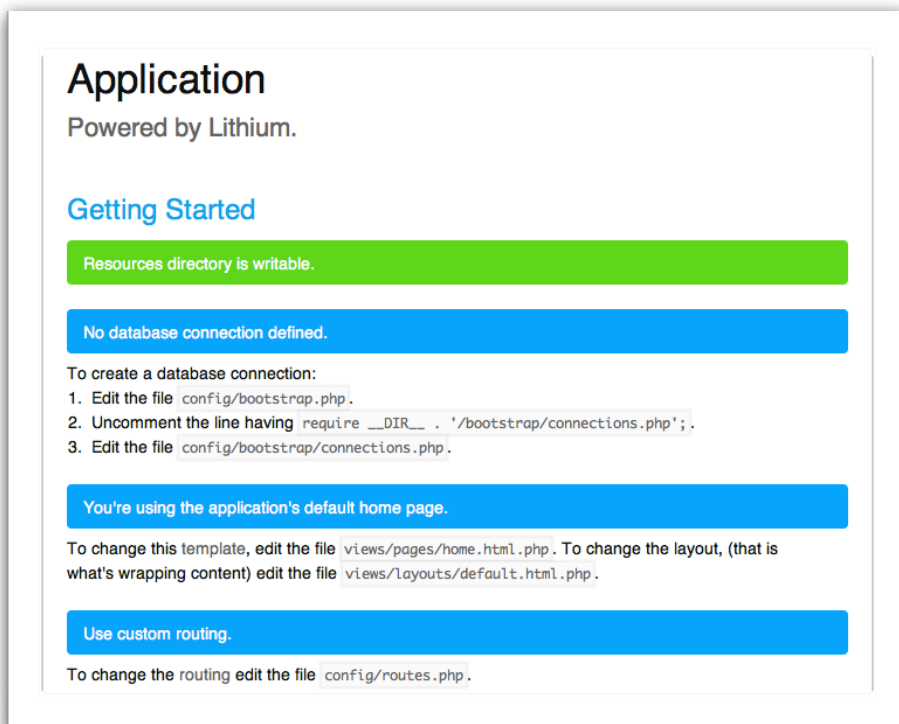
Now when we call `http://localhost/tutorial` it will fire the `config/bootstrap.php` file. But when this bootstrap file runs, it will try to require the Lithium core and will fail because Lithium is now located in a non-default location.

To fix this, open `tutorial_app/config/bootstrap/libraries.php` and change the `LITHIUM_LIBRARY_PATH` definition around line 62:

```
62 define( 'LITHIUM_LIBRARY_PATH', '/usr/lib' );
```

All of the necessary bootstrappings are in place. Launch `tutorial` in the browser and you'll now see the Lithium welcome screen, as shown in Figure 1.

FIGURE 1. The Lithium welcome screen.



Lithium is so good, it even steals my thunder. I can't even have you write a hello world method because it has already done it for you. You can test the Lithium routing engine right now by appending `helloworld` to the application's URL. Lo and behold, Lithium says "Hello World!" (If the Lithium developers were really clever, they'd plant a video of Rick Astley in there, but one can dream, right?)

You may have noticed that this welcome screen is reminding us that we haven't provided a data source yet. Just go to the `config/bootstrap/connections.php` file and uncomment the configuration for the data source you'll be using. The documentation there is self-explanatory: just populate the configuration array with the settings that match your database. Refresh the welcome screen, and you should see the previous `No database connection defined` line light up green, provided your database is working and your settings are correct.

Your First App: Not a Blog

Let's build something now that Lithium is up and running. Question sites are all the rage these days; gone are user forums and message boards. Let's make one for the Lithium platform that will give the fellas at Stack Overflow a run for their money. (Ahem. Yeah, right.)

Go to your app's `controllers` folder and create a file called `QuestionBoardController.php`. Go inside and name that class `QuestionBoardController` and have it extend `\Lithium\Action\Controller`.

Wait a second... Given that long introductory discussion about Lithium, you should have had some red flags go up. Did you get a sour taste in your mouth just now, or did you actually create the `QuestionBoardController.php` file? If you did create the file, let that be my first slap on your wrist. (You're now taking slaps on the wrist from me and perhaps preparing to sabotage my online reputation with a barrage of comments as payback, wondering why I could be so insulting so early on in your learning...)

This is why. Going this route would have violated a core paradigm of Lithium, and in principle, would steer us away from the robustness of the framework that makes Lithium worth using in the first place. Having one controller named after the app and set up to act as the main hub of our scripting logic consolidates a major portion of the app's concerns and processes into one place. Not good. Before long, `QuestionBoardController` would begin to resemble a simplistic library filled with service functions. This only makes sense in the event that the application-in-process is very basic and won't do much beyond one controller. For this kind of a project that will incorporate multiple and variegated methods, we will want to better leverage Lithium's intelligence and its robustness. We do this by adhering to its architecture paradigm.

This will require some theory before we even begin creating classes. What is our application going to do? What are its core concerns? How are these concerns interrelated? How will

these concerns interact with each other as distinct, semantic entities? How will they behave across class objects?

Obviously we can't answer all of these questions at so early a stage. But we can begin on the right trajectory by asking them. This keeps the project in line as more modules are created and attached here and there across the application.

A temptation is to start at the data level. We could begin designing the application's structure with the kinds of data classes that make up a Q&A website. People ask *questions* and give *answers*, so it's natural to go make tables for questions and answers, work up some models, and crank out a simple Q&A view to display the dynamic content. We will encounter these data-structural designs soon enough. But let's first tackle how to design the *concerns* of the application. These concerns will cut across the objects and the application, and may potentially demand some interoperability between modules. We wouldn't want to close off functionality by building up, even if unintentionally, coded walls.

By starting at the level of concerns, we quickly notice a different structure to the application take shape. An easier way of approaching this is to think of the grammar at play. At the data level of a Q&A website, there are nouns: questions and answers. At the concern level, there are verbs: asking and answering. *Asking* cuts across several objects in the system, and will encompass more than just the *question* data object. *Answering* will behave similarly. Many Q&A sites use reputation and activity levels to better achieve credibility for given answers, which thankfully, helps reduce low-quality responses that the Internet fosters with its incomprehensibly massive range of users. So this application may include *voting*, *ranking*, and *participating* concerns as well.

Still with me? If I've lost you, then let's back up a little bit. In essence the point that I'm driving is a *starting point* that differs from how a lot of developers do things. Notice how I haven't begun to write any code and yet I'm discussing all of these obtuse concepts that sound more fitting for a university lecture on theoretical systems design than a beginner's lesson on a PHP framework.

But that's just it: a how-to for *aspect oriented programming* (or AOP) is the beginner's lesson for *this* PHP framework. If you can absorb and understand this paradigm, all the rest is crumb cake. I'm taking the time now to introduce you to this paradigm so that the rest of your experiences with Lithium coincide with the code you'll be working with. Lithium code is an exceptional piece of AOP code, not just another PHP framework. And it has some significant benefits that are worth learning and considering.

The "Asking" Aspect and Its Dimensions

Here's the roadmap for the rest of this tutorial. I will lay out one aspect, what I will refer to as the Asking concern, and use that as an entrée to building the various pieces that make up a basic Lithium application. For the sake of efficiency, I will try to cut to the chase, and will

leave the kind of in-depth discussion I've already initiated for later. You've had enough, I'm sure, of the theory and now want to cut your teeth into some code.

As I lay out the Asking concern, I will specify some files to create using the convention that all files I mention are assumed to be following the application's root. If you're sticking with the Tutorial application installation I outlined above, then this would be the `tutorial` folder. In the case of the standard installation folder that comes off the Lithium repository, the application root is the `framework/app` directory. I have reconfigured the `tutorial` application to now be `qa`. From now on, I'll refer to the `qa` application in all of my examples.

When considering the Asking concern of the Q&A application, a few possibilities become apparent. First off, a user will initiate the Asking concern when he or she asks a question. Simple enough. What else, then, will stem from this action? The most obvious task will be saving that question to the database. This aspect of saving to the database falls within the expected line of operability that extends from the initial user action, which is to say that the probability that this line of operability will cause problems as the application code base grows or as additional features are added is low. Along the trajectory of the user starting an action and everything that the application will be called upon to do thereafter, the *saving* point will probably remain the closest in proximity to the starting point.

Let's make a note of that: the Asking sequence starts with the user asking a question and most likely will immediately follow with a save query to the database. No matter how we extend the code in the future, we can bet on the save query needing to stay at or very near step two in the sequence.

Saving the Question

The user will need a location where the site will provide the interface for asking questions. This location could be anything: `http://localhost/qa/ask` or `http://localhost/qa/ask-a-question`. The URL doesn't really matter because of how Lithium handles *routing*. A lot happens in the request cycle that makes it possible for Lithium to serve up all sorts of media types, from RSS feeds to JSON files to HTML or even non-web formats like DOC or RTF or PDF, provided you configure and build the requisite libraries for it to use. Lithium has abstracted out the route so that it can be mapped to whatever you need the URL to call.

When thinking of this principal location, it's good to go to that data layer I talked about earlier, the one that is more like the representation of nouns as opposed to verbs in the system. Of this layer, we ask: What will the user do? Well, ask questions. So the data layer will have in it a whole bunch of questions. The application needs a class to handle that data and another class to think about and direct that data to where it needs to go. The data handler is what is called the "model" and the data director is called the "controller." If you look in your Lithium app directory, you'll see a folder for each: `/controllers` and `/models`.

Create a new file in the `/models` folder called `Questions.php`. Place the following code in it:

```
1  <?php
2
3  namespace app\models;
4
5  class Questions extends \lithium\data\Model {
6
7  }
8
9  ?>
```

Line 3 declares the namespace for this `Questions` model class, which follows a Lithium convention: handily, it follows the folder location relative to the application root where the current file is housed. Per PHP syntax rules, a backslash is used in place of a forward slash for namespaces. So think of this namespace as corresponding to where the `Questions.php` file is located. You won't need to rename the `app` portion of the namespace if you have renamed the parent application folder—this actually matches with a method called in the `/config/bootstrap/libraries.php` file that defines `app`. You would only have to adjust the namespace if you tinkered with that file, which at this point we haven't done, so we'll stick with `app\models` for the namespace.

Line 5 names the class `Questions` and extends it from `\lithium\data\Model`, which is a class in the Lithium core. This, too, matches the namespace convention in Lithium. Because of how Lithium lazy-loads its core classes, any class in the application root can extend from any class in the Lithium core as long as the full namespace is used in the `extends` declaration. Say you wanted to extend this class from the `StaticObject` class located in the `lithium/core` folder instead. Easy enough. You would just replace `\lithium\data\Model` with `\lithium\core\StaticObject`. No need for `require` or `include` statements.

The presence of the `Questions` class now means that the data the user will provide in the form of questions has an interface between it and the database. You'll see soon enough how easy it will be to perform all kinds of useful data handling functions on these questions data without any more than what we've written here in `/models/Questions.php`.

Create a table in the connected database called `questions`. Give it an `id` integer field that auto-increments and another field named `text` with the `TEXT` type. The database is now ready to receive questions.

With the class responsible for handling the data in place, now the application needs a director, what we've already identified as the controller. Create the `/controllers/QuestionsController.php` file and give it the following code; it will bear some resemblance to what you just did with the model class:

```
1  <?php
2
3  namespace app\controllers;
4
5  use app\models\Questions;
```

```
6
7   class QuestionsController extends \lithium\action\Controller {
8
9     public function ask() {
10        if (isset($this->request->data[ 'text ' ])) {
11            $response = false;
12            $question = Questions::create($this->request->data);
13
14            if ($question->save()) {
15                $response = true;
16            }
17            return $response;
18        }
19    }
20 }
21
22 ?>
```

Notice on line 3 how this namespace corresponds to the `/controllers` directory where you've created the `QuestionsController.php` file. Line 5 brings in the `Questions` model class with the use declaration. Now, whenever you wish to call that class in the `QuestionsController`, you won't have to use the entire namespace. Line 12 demonstrates this in action: the `create` method is run on the `Questions` class, and you didn't have to type out the full `\app\models\Questions::create()` thanks to line 5 (although you could have and it would work, but a tad redundant).

I'll explain what this code does in a minute. But first let's pause and have you take stock of where you're at in the development process. You've set out to implement the Asking concern or aspect into your brand-new application by figuring out the starting gate for that aspect, which was that a user asks a question. You then observed that an obvious action needed to follow that initial user action, which was to save that question to the database. You learned that Lithium provides a model structure for handling the data and a controller structure for directing that data. You just created corresponding classes in the `Questions` and `QuestionsController` objects.

Now when a user calls `http://localhost/qa/questions/ask`, Lithium's router will by default map that URL to the `QuestionsController` and its `ask` method, or to put it more technically, it will map to `\app\controllers\QuestionsController::ask()`. You have created a matching controller class and method, so that URL should call the right mechanism in the application to handle the Asking aspect as we have thus far implemented it.

There's just one minor issue at this stage—the user has nothing with which to interact yet. The controller layer happens between visual interfaces, and the model layer happens between controller and other actionable interfaces in the application, so as of yet, the user won't see anything. Furthermore, Lithium will throw an exception if you try to call the URL pointing to the newly written `QuestionsController::ask()` method, saying that it couldn't locate a matching “view” file. Before the Asking aspect can be initiated by a user action, there needs to be an interface for that user to actually put the Asking into motion. With the

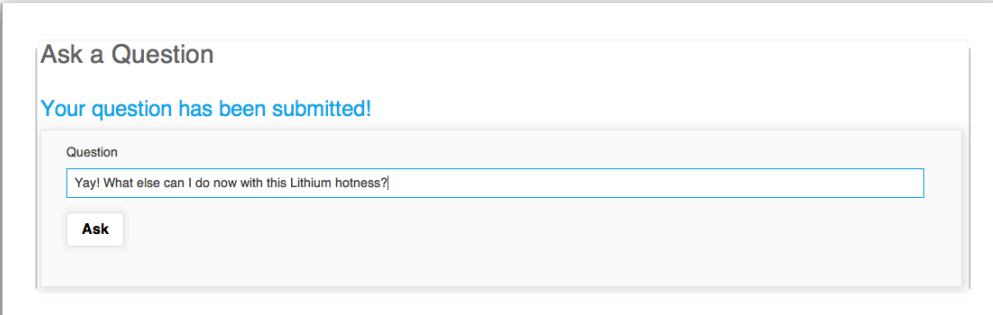
controller and model layers in place, all that is left is to provide the presentation layer. We do this by creating a view file in our application.

Create that view file by writing the following code to `/views/questions/ask.html.php`. You will have to create the `questions` folder inside of the `views` folder first.

```
1 <h2>Ask a Question</h2>
2 <?php if (isset($response) && $response == true): ?>
3     <h3>Your question has been submitted!</h3>
4 <?php endif; ?>
5 <?=$this->form->create();?>
6     <?=$this->form->field( ' text ' );?>
7     <?=$this->form->submit( ' Ask ' );?>
8 <?=$this->form->end();?>
```

When the user initiates `QuestionsController::ask()` through a matching URL, the function will first check to see if any form data has been submitted to it by looking to the controller's POST data container, `$this->request->data`. Because the user will not have interfaced with the view before, this will turn up empty, and so nothing will be done except call the default view, which is in this case going to correspond with the file you just created as `/views/questions/ask.html.php`. With this visual interface in view, the user be presented with a text field (see line 6) and a submit button (line 7). You have called a handy form class known as the Form helper that will be automatically loaded as Lithium renders the view. This Form helper generates the HTML form elements on lines 5 and 8 that wrap the field and submit button in the necessary form. It also supplies the route pointing back, by default, to `QuestionsController::ask()`, which of course can be changed by supplying some parameters to the `Form::create()` function.

The first two points of the Asking concern are now in place: the ability for the user to ask a question and the saving mechanism for getting that question into the database for distributed use. Go ahead and fire up `http://localhost/qa/questions/ask` and use the form.



Ask a Question

Your question has been submitted!

Question

Yay! What else can I do now with this Lithium hotness?

Ask

Other Concerns Related to the Asking Aspect

Remember all that talk about concerns and aspects? That theory lesson wasn't for nothing. At this stage we begin to ask what other aspects stem from a user Asking a question. This will include a high degree of variability due to the unique set of features this application will support, and that can include anything. Perhaps this Q&A site will assign the question to potential responders. Maybe it will optimize the question content for search engines. It might sort and rank the questions based on relevance or larger traffic trends or within a category. If we're not careful, we could obstruct those future features by keeping everything tied down in one sequence of logic. Here's an illustration.

Imagine that the client of death has seen your killer Q&A app but wants to scrap a handful of features and add something truly insane, like a complete online university, because, come on, *that's* the way to charge fifty grand to field questions. In that scenario, what started out as a Q&A service has morphed into an entirely different architecture. With poor architecture design at the outset, you would likely have to rewrite the program from scratch. But if you keep the code flexible, it's not far fetched to say that you could probably keep extending and adding away, and the existing code could cope with the changes. At least, this would be the goal.

I've gone to this extreme for effect, so I recognize that this is probably unrealistic. But on a lesser scale this very thing happens all the time—in fact, I daresay that it is a truism in practical computer programming that change to the code base is constant in almost any project. The question is one of degrees, not maybe; of when, not if.