

The Facade Design Pattern

brian d foy, comdog@panix.com

Abstract

The Facade design pattern provides an easy-to-use interface to an otherwise complicated collection of interfaces or subsystems. It makes things easier by hiding the details of its implementation.

1 Introduction

The Facade design pattern connects the code we write for applications, which do specific tasks, such as creating a report, and the low level implementation that handle the details, such as reading file, interacting with the network, and creating output. The facade is an interface that an application can use to get things done without worrying about the details. The facade decouples these layers so that they don't depend on each other, which makes each easier to develop, easier to use, and promotes code re-use.

I can use this design pattern to deal with a complex system that already exists, or one that I want to make from scratch. Several Perl modules available on the Comprehensive Perl Archive Network (CPAN) represent facades, even if they do not admit it.

2 Illustration of use

To request a simple file from a web site, I have to create a connection to the web site, request the resource using a proper HyperText Transfer Protocol (HTTP) request, receive the HTTP response, parse the response, and finally handle the data. I have to do much more work if I want to handle common web features like cookies, forms, and caching. If I want to fetch a resource from an FTP server instead of a web server, I need to handle a completely different protocol.

If I look at this problem, some immediate objects present themselves: connection, request, response, and resource. However, I only want to fetch the resource and continue on with my real work rather than deal with myriad objects to do something that is logically so simple. To code this myself in a reasonable amount of time might take a couple of screenfuls of code depending on how careful I am and how many features I decide to support.

The LWP module (Library for WWW in Perl) provides a facade for doing all of these things. I tell LWP to fetch a resource and it does the rest, including all of the protocol-specific details for HTTP, FTP, or any other protocol that LWP understands.

In code listing 1, I use `LWP::Simple` which makes fetching a web resource as simple as it can get. I do not have to specify the protocol, the connection method, or parse the response. Indeed, if I know the URL, and I can fetch the resource.

Code Listing 1: A web facade

```
1 use LWP::Simple qw(get);
2
3 my $url = 'http://www.perl.org';
4 my $data = get( $url );
```

The `LWP::Simple` module is a facade—it provides a simple interface that unifies protocol, network, and parsing aspects of the problem so that I do one thing—fetch the resource. If someone changes `LWP` or underlying implementations, I do not have to change my script and I still benefit from the improvement.

3 Focus on the task

A facade restricts the functionality, and thus the complexity, of a system by creating specialized interfaces for specific tasks. The `HTML::Parser` module is a base class, so the programmer must write a subclass that tells the parser what to do and when to do it, but it does not perform a specific task other than the parsing, such as syntax checking or data extraction. However, as in my `LWP` example, I simply want to complete a single, logical task—not program `HTML::Parser` subclasses. In reality, I like writing `HTML::Parser` subclasses, but not everyone with whom I work does, so I can create facades for them.

Facades to `HTML::Parser` do small, common tasks while they hide all of the complexity behind-the-scenes. I am the only one in the programming team who has to understand `HTML::Parser` so I can create much simpler interfaces for the rest of the team. They should not have to write an entire subclass if they only want to extract the links of an HTML document, for instance. Simple things should be simple.

If I wanted to extract references (which most people call “links”) from an HTML document, I can use the `HTML::LinkExtor` module. It handles most of the complexity for me while still giving me a reasonable amount of flexibility through a callback mechanism. In code listing 2, I use a callback to extract all of the anchor references (the `HREF` attribute from the `A` tag). I still have to do a bit of the dirty work since `HTML::LinkExtor` passes the tag name and a list of attribute–value pairs to `call_back()`. I still have to know the details of the implementation of `HTML::LinkExtor` to work with it.

Code Listing 2: `HTML::LinkExtor`

```
1 require HTML::LinkExtor;
2
3 use vars qw( @links );
4
5 sub call_back
6 {
7     my( $tag, %attr ) = @_;
8     return unless exists $attr{href};
9
10    push @links, $attr{href};
11 }
12
13 my $parser = HTML::LinkExtor->new( \%call_back, "http://www.example.com" );
14
15 $parser->parse_file("index.html");
```

I can sacrifice flexibility for convenience by using a simpler Facade, `HTML::SimpleLinkExtor`, which simply returns the references but does not have a callback mechanism. In code listing 3, I do the same thing that I do with `HTML::LinkExtor` example in code listing 2, and my fellow programmers do not know how `HTML::SimpleLinkExtor` does it. They do not need to worry about writing the callback function. They simply get the result that they need.

```
_____ Code Listing 3: HTML::SimpleLinkExtor, just HREFs _____  
1 use HTML::SimpleLinkExtor;  
2  
3 my $extor = HTML::SimpleLinkExtor->new( "http://www.example.com" );  
4 $extor->parse_file("index.html");  
5  
6 my @links = $extor->href;
```

If I want to do something different with `HTML::LinkExtor`, like extracting URLs from tags with SRC attributes, I have to modify the `callback` subroutine, or I can use a method from `HTML::SimpleLinkExtor`, a simpler facade, like I do in code listing 4.

```
_____ Code Listing 4: HTML::SimpleLinkExtor, all links _____  
1 use HTML::SimpleLinkExtor;  
2  
3 my $extor = HTML::SimpleLinkExtor->new( "http://www.example.com" );  
4 $extor->parse_file("index.html");  
5  
6 my @all_links = $extor->links;
```

In both of these examples, the facades allows programmers to focus on the task—extracting links—rather than on the programming. Since each facade provides a task-oriented interface, programmers do not spend time thinking about how the task should be completed, just as they do not think too much about HTTP or TCP/IP when they use their web browser to visit their favorite web pages.

The `HTML::SimpleLinkExtor` works for most uses, but also cannot handle complex cases at all. Reduced flexibility is the major consequences of a facade. The more restrictive facades are easier to use at the cost of flexibility. `HTML::LinkExtor` has more flexibility, but is a bit more complicated and I have to do more work to use it. The more flexible interfaces can handle more situations and respond to special cases at the cost of simplicity. Specific situations require different levels of flexibility and simplicity, and as a result, different facades, if any at all.

4 Facades promote reusability

Many programmers already use a sort of facade, although they typically call it a subroutine. Subroutines did not always exist, and they represented a pattern of their own at one time. Today most languages take subroutines for granted even though they are the foundation of re-usable code.

The abstract nature of the subroutine allows programmers not only to group program statements into logical operations behind a subroutine name, like the facades in the previous section, but they also allow

programmers to reuse that group of program statements without repeatedly typing them. Programmers can reuse and share collections of subroutines that they group in libraries which can form the interface of a facade.

What if I want to check the status of a web resource? If I went through all of the steps myself every time I needed to do this, I would have to create an HTTP request, connect to the web server, receive the response, parse the response, and check the response code against known status codes. Later, once I have coded this four or five lines of LWP code (or 15 to 20 lines of socket code) in all of my applications that need it, I will probably discover that I need to fix some of the code and to make the change in several places. If I put all of that code in a subroutine that all of my applications can share, I only have to fix things in one place and programmers using my module do not need to do anything at all.

I first ran into the link validation problem when I created a user-configurable directory of internet resources which I wanted to validate every day. I wanted to make sure that the links in the directories actually led to a web page rather than the annoying “404 Not Found” server error. I also wanted to catch dead links before a customer added them to his directory. I needed to do the same simple task in several applications, and the few lines of repeated code in each application seemed so innocent that I missed the obvious refactoring potential.

With a couple of customers using the service, I did not notice anything amiss with my validation code. It caught dead links and did not have false positives. With tens of users and a couple hundred thousand resources to validate, I discovered that not all web servers respond in the same way to certain types of HTTP requests, and that some servers even had little known bugs. One notorious server returned an HTTP error for any HEAD request¹, so I had to program some special cases. The task which I thought was simple grew much more complex, but I wanted to work on the level of a “ping”—a simple “yes” or “no” answer. Tests on small data sets did not reveal any problems in the parts per ten thousand range, but things quickly got out of hand after that.

My refactored solution was a facade. At the application level I did not care about server eccentricities, work-arounds for HTTP non-compliance, or most error recovery. I simply wanted to know if the URL actually pointed to something. I created a glorified subroutine, gave it a module name, and used it whenever I needed an HTTP response code. I uploaded the module to the Comprehensive Perl Archive Network (CPAN) as HTTP::SimpleLinkChecker. Code listing 5 shows the entire facade—a single function behind which all of the real work takes place. The facade takes care of all of the details, including all of my accrued knowledge about specific server behaviors, so that it could recognize possible problems and double check errors to a HEAD request by actually downloading the resource.

Code Listing 5: HTTP::SimpleLinkChecker

```
1 use HTTP::SimpleLinkChecker qw(check_link);
2
3 my $code = check_link("http://www.example.com");
```

If someone uses this module and decides to upgrade to a newer version when I release one, he can get the benefit of all of my improvements and enhancements without changing any of his code, while at the same time, he benefits from any enhancements to its LWP infrastructure even if he does not use the latest version of HTTP::SimpleLinkChecker. This *loose coupling* makes a programmer’s life much easier since the facade hides changes to the underlying system. Changes in other parts of the system have little or no maintenance consequences on the programmer’s application. Since the facade depends very loosely on the underlying

¹Every HTTP request specifies a method. A HEAD request asks the server for the resource’s meta data, but not the resource itself so it does not have to download potentially large amounts of data.

implementation, I can distribute it separately. Other people do not need a specific version of LWP. I make it easy for people to use so that they will use it rather than going through the pain and suffering that I did.

5 Facades as objects

A facade object acts as the gatekeeper of all method calls for the underlying implementation. Each method may in-turn act upon additional objects or classes to perform its task, but the programmer does not have to know anything about that. The facade object knows which underlying objects handles the real work and delegates parts of each task appropriately².

If I want to parse an HTML page to extract various things from the <HEAD> as well as some of the links, I can use HTML::HeadParser and HTML::LinkExtor, but at the application level that is too much detail. I am stuck thinking about HTML parsing when I should be getting my real work done. I can also create my own HTML facade that hides the two modules—or any other implementation—that I use.

In code listing 6, I wrap the interfaces to HTML::HeadParser and HTML::SimpleLinkExtor in a single interface so I only have to deal with all of them in my application.

```

_____ Code Listing 6: The HTML::SimpleExtractor facade _____
1  package HTML::SimpleExtractor;
2
3  require HTML::HeadParser;
4  require HTML::SimpleLinkExtor;
5
6  sub new
7      {
8      my( $class, $url ) = @_;
9
10     return unless $data = LWP::Simple->get($url);
11
12     bless \$data, $class;
13     }
14
15  sub title
16      {
17      return HTML::HeadParser->new()->parse( $_[0] )->header('Title');
18      }
19
20  sub links
21      {
22      HTML::SimpleLinkExtor->new()->parse( $_[0] )->links;
23      }
24
25  1;
26
27  __END__

```

²Delegation represents another sort of pattern with which the Facade pattern might collaborate. Many patterns work in concert with other patterns, and although I do not discuss Delegation here, you can read about it in the documentation of Damian Conway's Class::Delegation or Kurt Starsinic's Class::Delegate

Once I have my facade in place, I can write applications that I do not need to couple to any particular module. Since my program in code listing 7 does not know that I used `HTML::HeadParser` it does not need to change if I decide to use something different for that portion of the task. The facade hides the changes.

Code Listing 7: `HTML::SimpleExtractor`

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  use HTML::SimpleExtractor;
5
6  my $html = HTML::SimpleExtractor->new('http://www.example.com');
7
8  my $title = $html->title;
9  my $links = $html->links;
10
11 $ = "\n\t";
12
13 print "$title\n\t@links\n";
14
15 __END__
```

In this case, I can change any of the details involved with fetching and parsing the HTML to something smarter and more efficient later. This can be quite expedient when the responsibility for the facade and the application belong to different programmers or teams, or when it would take much longer to fully implement the facade than to finish any of the applications. I can create something that works today even if it is not the best implementation then I can incrementally change and improve it as time allows.

Even though my example is very simple-minded, I get the job done. I hide the two modules behind the facade, and I can immediately use `HTML::SimpleExtractor` in my applications. Once I have more time to devote to the module, I can change how it does its job while all of my applications that use it stay the same. None of my application code depends on the specifics of the implementation, and may not even know that the implementation has changed.

6 Ad hoc facades

Most frequently the work programmers seem to do involves an established base code that has entrenched itself into the work flow of their organizations, and the further away they are from the creation time of this code, the more difficult it is to maintain or learn, especially if it is as sparsely documented as most such code I have seen. New team members can have an especially difficult time learning a byzantine code base which ends up strangling the work flow.

A facade can gradually fix this without an immediate or complete rewrite of the old code. Since a facade provides a unified interface to an complex, underlying system, it can also hide years of improvements, multitudes of styles, and unforeseen problems in the original code. A new interface that connects various legacy subsystems of the old code provides a way for programmers to replace the functionality later while creating new applications with the new interface. New programmers do not need to learn the entire system if one of the old salt programmers create a facade for them.

As more and more applications use the new facade, and hopefully fewer and fewer applications use the old

code base as programmers gradually replace them, the application base moves towards something much more maintainable since the application code does not rely on the underlying facade implementation. Programmers can re-implement portions of that at their leisure without breaking applications.

Typically, these sorts of facades pull together a particular way of doing things in a particular context such as a special business need or workflow. I might have several closely related applications, and as I develop them in parallel parts of them start to look the same because they do similar things and use the same resources. Such an application's first few lines might look like code listing 8 which uses several Perl workhorse modules.

Code Listing 8: Using several modules

```
1  #!/usr/bin/perl -w
2  use strict;
3
4  require cgi-lib.pl;
5  use DBI;
6  use HTML::Parser;
7  use HTML::TreeBuilder;
8  use LWP;
9  use Text::Template;
10
11 # my Perl implementation here
12 __END__
```

I can refactor those applications and use a facade to contain all of the details about how I do the work. I want the facade to represent the task, not the method. If I use certain modules by local policy, then I only have to enforce that policy behind the facade rather than in every application. For this suite of hypothetical applications I create a module I call Tsunami³—the name I give my fictional product. Instead of all of the modules I use in code listing 8, including the notoriously old `cgi-lib.pl`, I only have to use one module, as in code listing 9. This also means that other team members, by policy if not practice, only use one module too. If, for some reason, policy changes so that Tsunami should use `CGI.pm` instead of `cgi-lib.pl`, I only have to change one file. If I improve Tsunami, everybody benefits.

Code Listing 9: Using one module

```
1  #!/usr/bin/perl
2
3  use Tsunami;
4
5  my $wave = Tsunami->new(...);
6
7  $wave->fetch('http://www.example.com');
8
9  my $title = $wave->title();
10
11 my $txt   = $wave->as_text();
12
13 print $txt;
14
15 __END__
```

³a deluge of code

7 A priori facades

If I have the luxury of prior thought and planning, and I know that some parts of the system resist planning, I can use a facade to present an application programming interface, and build up the rest of the stuff as I find out more and more about the problem.

Once I go through the object-oriented analysis process and have identified the objects, I can also identify the different ways that the programmers will use those objects. For instance, if I want to create an application to send messages between two computers, I know that I need a network object and a message object. These objects make it easy to deal with related sets of information my program must maintain.

I want to create a application that can “chat” with another, meaning that the two applications can send messages back and forth between each other. I can use a facade to represent a simple interface, with `send()` and `receive()` methods. I might need more later, but in this contrived example I pretend that I do not know everything this application might have to do because the specifications are fuzzy and the scope of the problem scares the project planners (in this case, me).

When I start programming, nothing works because I have not written any code to represent the objects, and at that level I need all of the objects for the other ones to do their part. Since a facade hides these objects, it also hides their absence as well. I can get something in place quickly, just as in my `HTML::SimpleExtractor` example in code listing 6, and be on my way.

As with all new programming I start, the first thing I do is write a test suite. Since I have no objects to test, all of the tests should fail, which is my first real test—tests fail when they should.

I create my module workspace with `h2xs`, which creates a `t` directory for test files⁴. In my test file I add the test in code listing 10. Since the `send()` method should die with an internal error message, I check to see that it does. So far the module `Chat.pm` is the template that `h2xs` created for me.

Code Listing 10: Test an unimplemented method

```
1 eval {
2     Chat->send('Come here Mr. Watson, I need you');
3 };
4 print $@ ? 'not ' : '', "ok\n";
```

One step beyond that I want to test that the parts of the interface exist, so I need an interface. I need to create the `Chat.pm` module. In code listing 11 I have a minimal module which has one class method, `send()`. I have not really implemented it yet, so I call the `die()` function if a program calls the method.

Code Listing 11: `Chat.pm`

```
1 package Chat;
2
3 use vars qw( $UNIMPLEMENTED );
4 $UNIMPLEMENTED = "Not implemented!";
5
6 sub send    { die $UNIMPLEMENTED }
7
8 1;
```

⁴see `Test::Harness` for details about testing

I expect the test from code listing 10 to fail because `send()` calls the `die()` function, but I can modify the test to see if I get the right message in `$@`. In code listing 12, the test succeeds even though the `send()` uses `die()`, since that is the behavior I expect—part of the interface now exists.

Code Listing 12: Test an unimplemented method

```
1 eval {
2     Chat->send('Come here Mr. Watson, I need you');
3 };
4 print "$@" eq "$Chat::UNIMPLEMENTED ? ' ' : 'not ', "ok\n";
```

A similar test for an undefined method should give me a different sort of error. In code listing 13 I test to see if the `receive()` method exists, and if it does, then something is wrong. I have not defined `receive()` yet.

Code Listing 13: Test a non-existent method

```
1 eval {
2     Chat->receive('I am on my way');
3 };
4 print "$@" ? ' ' : 'not ', "ok\n";
```

Although I still do not have any objects, I can change my `send()` method to take arguments. In code listing 14 the `send()` method takes a message and a recipient argument, although I have not said anything about what they are. I have, however, made progress on the interface even though I still do not have anything to actually do the work.

Code Listing 14: `Chat::send()` with arguments

```
1 sub send
2     {
3     my( $class, $message, $recipient ) = @_;
4
5     return unless defined $message and defined $recipient;
6
7     return 1;
8     }
```

My test for `send()` changes to make sure it does the right thing for different argument lists. In code listing 15 I add three tests for different numbers of arguments. Only the call to `send()` with the right number of arguments should succeed. The other tests check for failure when `send()` should fail.

Code Listing 15: `send()` with different numbers of arguments

```
1 # should fail -- no recipient
2 eval {
3     Chat->send('Come here Mr. Watson, I need you');
4 };
5 defined "$@" ? not_ok() : ok();
6
```

```
7 # should fail -- no message or recipient
8 eval {
9     Chat->send();
10 };
11 defined $$ ? not_ok() : ok();
12
13 # should succeed
14 eval {
15     Chat->send('Come here', 'Mr. Watson');
16 };
17 defined $$ ? not_ok() : ok();
```

This process continues as I add more to the interface and as I implement the objects that will actually do the work. In the mean time, I have done useful work that has gotten me towards my goal, and I have created a suite of tests to help me along the way. The implementation does not concern me too much at this point because I can easily change it later. The rest of the project depends on the facade. Only the facade knows about the implementation, so everything else, including applications and tests, do not have to wait for the complete implementation to start work.

Once I progress far enough to have Message and Recipient objects, if I decide I need them, I can change my send() method to use them. In code listing 16 I check each argument to ensure that they belong to the proper class. Each object automatically inherits the isa() method from UNIVERSAL, the base class of all Perl classes, and returns TRUE if the object is an instance of the class named as the argument, or a class that inherits from it. All of my tests still do the same thing.

Code Listing 16: Argument checking

```
1 sub send
2     {
3     my( $class, $message, $recipient ) = @_;
4
5     return unless $message->isa('Chat::Message') and
6         $recipient->isa('Chat::Recipient');
7
8     return 1;
9     }
```

If later I change the objects or their behavior, I have not wasted too much time. My facade and its tests still work. Any application I have written does not need to change significantly. The facade handles the details and the interactions between the various objects. At the same time, other programmers can start to use the interface to create applications. The programs will not work until everything is complete, of course, but the programmers have a jump start on the process because they can code and test before everything is in place. They essentially work in parallel, instead of serially, with the programmers implementing the objects and the facade. The creation of classes is not a work flow bottleneck since the facade decouples the application and lower level implementations.

8 Perl modules which are facades

Several modules exhibit a Facade design pattern, although their authors may not have thought about design patterns or facades in particular. A good design stays out of the way and does not draw attention to itself. A good facade keeps most of us truly ignorant about whatever is behind it.

Most of the modules on CPAN with “Simple” in their name use the facade design pattern, including LWP::Simple, HTTP::SimpleLinkChecker, and HTML::SimpleLinkExtor which I used for examples.

8.1 LWP

The LWP family of modules act as a facade with a unified interface to various network protocols including HTTP, HTTPS, FTP, NNTP, and even the local filesystem. It can handle some protocol specific details too.

8.2 DBI

The DBI module provide a simple interface to several database servers or file formats. I can query several types of server or file formats with the same interface while DBI—actually the appropriate DBD—handles the connection, query, response, and other tasks. The DBI interface even allows me to change the database server behind the scenes (from SQL Server to postgresql, perhaps) without changing much more than the DBI->connect statement. I do not need to worry about the connection implementation, protocols, or data format.

8.3 Tied classes

Perl’s tie functionality is a type of facade. What looks like a normal Perl scalar, array, or hash stands-in for a possibly complex object behind the scenes. The most impressive use of this sort of facade, in my opinion, is the Win32::TieRegistry module, which represents the quite complex Microsoft Windows registry as a tied hash. This module even sees the following as equivalent:

```
\$Registry->{'LMachine/Software/Perl'}  
\$Registry->{'LMachine'}->{'Software'}->{'Perl'}
```

I do not have to know very much, if anything, about the Registry. I do need to know how I name a key, but I do not need to know how it is stored or accessed since I already understand Perl hashes and references. This module takes what I already know and lets me use it instead of learning low-level vendor interfaces.

This facade also allows me to develop on foreign platforms. I can reimplement the code so that I can have a Unix version of the Windows registry, for instance. This fake version of the Registry allows me to use all of the great tools and setups I already have on my unix accounts while targeting Windows platforms. If my application had to use explicit calls to the Windows programming interface I would not be able to do that.

9 Conclusion

The Facade design pattern provides a simple interface to a complex collection of modules or code. The Facade design pattern removes the details of the task from the application layer which allows me to focus on the task rather than the details, as well as allowing me to change implementations without changing my applications. This makes code more modular, maintainable, and easier to use. It decouples application code from the low level implementation which allows parallel development at different levels.

10 References

You can read more about design patterns in *Design Patterns*, Erich Gamma, Richard Helm, Ralph Johnson, Jon Vlissides, Addison Wesley, 1995.

Test suites for Perl modules usually use `Test::Harness`, although I find actual test files easier to understand. I have some simple tests in the `test.pl` of `HTML::SimpleLinkExtractor` distribution, or the `t` directory of `HTTP::SimpleLinkChecker`.

All real modules in this article are in the Comprehensive Perl Archive Network (CPAN)
<http://search.cpan.org>

11 About the author

brian d foy is the publisher of *The Perl Review* and the author of several modules available on CPAN, including `HTML::SimpleLinkExtor` and `HTTP::SimpleLinkChecker`. He teaches and writes on Perl topics in between doing real work.