From Procedural to Object Oriented PHP

Patkos Csaba on Jun 21st 2013 with 13 Comments

Tutorial Details

- **Difficulty**: Intermediate
- **Estimated Completion Time**: 60 minutes

This tutorial was inspired by a speech given by Robert C. Martin that I watched a year or so ago. The main subject of his talk is about the possibility of picking The Last Programming Language. He addresses topics such as why should such a language exist? And what it should look like? However, if you read between the lines, there was another interesting idea that caught my attention: the limitations that each programming paradigm imposes upon on us programmers. So before we get into how we could go about converting a procedural based PHP app into an object oriented one, I want to cover a little bit of theory beforehand.

Paradigm Limitations

So, each programming paradigm limits our ability to do whatever we want to do. Each of them takes something away and provides an alternative to achieve the same result. Modular programming takes away unlimited program size. It enforces the programmer to use maximum sized modules and each module ends with a “go-to” statement to another module. So the first limitation is upon size. Then, structured programming and procedural programming takes away the “go-to” statement and limits the programmer to sequence, selection and iteration. Sequences are variable assignments, selections are if-else decisions, and iterations are do-while loops. These are the building blocks of most programming languages and paradigms today.

Object oriented programming takes away pointers to functions and introduces polymorphism. PHP doesn’t use pointers in a way that C does, but a variant of these pointers to functions can be observed in Variable Functions. This allows a programmer to use the value of a variable as the name of a function, so that something like this can be achieved:

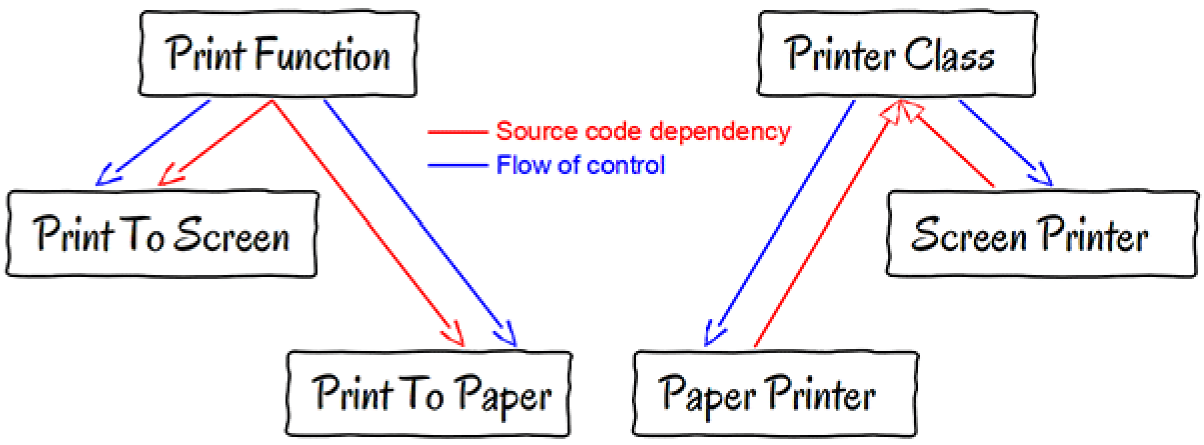
```
function foo() {
    echo "This is foo";
}

function bar($param) {
    echo "This is bar saying: $param";
}

$function = 'foo';
$function();         // Goes into foo()

$function = 'bar';
$function('test');   // Goes into bar()
```

This may not look important at first sight. But think about what we can achieve with such a powerful tool. We can send a variable as a parameter to a function and then let that function call the other one, referenced by the value of the parameter. This is amazing. It allows us to alter the functionality of a function without it knowing it. Without the function even noticing any difference.

We actually can do polymorphic calls with this technique as well.

Now, instead of thinking about what pointers to functions provide, think about how they work. Aren't they just hidden "go-to" statements? Actually, they are, or at least they are very similar to indirect "go-to"s. Which is not very good. What we have here is in fact a clever way to do "go-to" without using it directly. I have to admit that in PHP, as the example above shows, it is quite easy to understand, but it may get confusing with larger projects and many different functions passed from one function to another. In C it is even more obscure and hideously difficult to understand.

However, just taking away pointers to functions is not enough. Object oriented programming must provide a replacement, and it does, in an elegant way. It offers polymorphism with an easy syntax. And with polymorphism, comes the biggest value object oriented programming offers: The flow of control is opposed to the source code dependency.



In the image above we illustrated a simple example of how polymorphic calls happen in the two different paradigms. In procedural or structural programming, the flow of control is similar to the source code dependency. They both point toward the more concrete implementation of the printing behavior.
In object oriented programming, we can reverse the source code dependency and make it point toward the more abstract implementation, while keeping the flow of control pointing to the more concrete implementation. This is essential, because we want our control to go and reach the most possible concrete and volatile part of our code so that we can get our result exactly as we want it, but in our source code we want the exact opposite. In our source code we want the concrete and volatile stuff to stay out of the way, to be easy to change and to affect as little as possible of the rest of our code. Let the volatile parts change frequently but keep the more abstract parts unmodified. You can read more about the Dependency Inversion Principle in the original research paper written by Robert C. Martin.

The Task at Hand

In this chapter we will create a simple application to list out Google calendars and the events within them. First we will take a procedural approach, using only simple functions and avoiding any kind of classes or objects. The application will allow you to list your Google calendars and events. Then we will take the problem one step further by keeping our procedural code and starting to organize it by behavior. Finally we will transform it into an object oriented version.

PHP Google API Client

Google provides an API client for PHP. We will use it to connect to our Google account so that we can manipulate calendars there. If you want to run the code, you must set up your Google account to accept calendar queries.
Even though this is a requirement for the tutorial, it is not its main subject. So instead of me repeating the steps you have to take, I will point you to the right documentation. Don’t worry, it is very simple to setup and it takes only about five minutes.
The PHP Google API client code is included in each project from the example code attached to this tutorial. I recommend you use that one. Alternatively, if you are curious about how to install it yourself, check out the official documentation.
Then follow the instructions and fill in the information in the `apiAccess.php` file. This file will be required by both the procedural and object oriented examples, so you do not need to repeat it. I left my keys in there so you can more easily identify and fill in yours.
If you happen to use NetBeans, I left the project files in the folders containing the different examples. This way you can simply open the projects and run them immediately on a local PHP server (PHP 5.4 is required) by just simply selecting **Run / Run Project**.

The client library to connect to the Google API is object oriented. For the sake of our functional example, I wrote a small set of functions that wrap in them, the functionalities we need. This way, we can use a procedural layer written over the object oriented client library so that our code will not have to use objects. If you want to quickly test that your code and connection to the Google API is working, just use the code below as your `index.php` file. It should list all the calendars you have in your account. There should be at least one calendar with the **summary** field being your name. If you have a calendar with your contact's birthdays, that one may not work with this Google API, but don't panic, just choose another one.

```
require_once './google-api-php-client/src/Google_Client.php';
require_once './google-api-php-client/src/contrib/Google_CalendarService.php';
require_once __DIR__ . '/../apiAccess.php';
require_once './functins_google_api.php';
require_once './functions.php';
session_start();

$client = createClient();
if(!authenticate($client)) return;
listAllCalendars($client);
```

This `index.php` file will be the entry point to our application. We won't use a web framework or anything fancy. We will just simply output some HTML code.

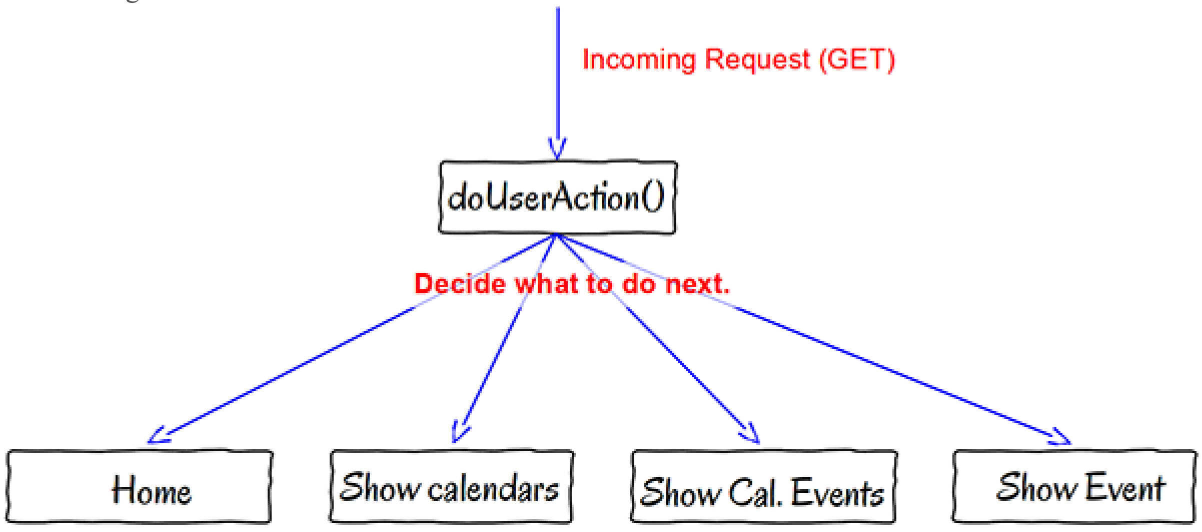
A Direct Procedural Approach

Now that we know what we're building and what we'll be using, go ahead and download the attached source code. I will provide snippets from it, but in order to see the whole thing, you'll want to have access to the original source.

For this approach, we just want to get things working. Our code will be organized in a very rudimentary way, with just a few files, like this:

- **index.php** – the only file we access directly from the browser and pass to it GET parameters.
- **functions_google_api.php** – the wrapper over the Google API we talked about above.
- **functions.php** – where everything happens.

`functions.php` will house everything that our application does. Both the routing logic, the presentations, and whatever values and behavior may be buried in there. This application is pretty simple, the main logic is as follows.

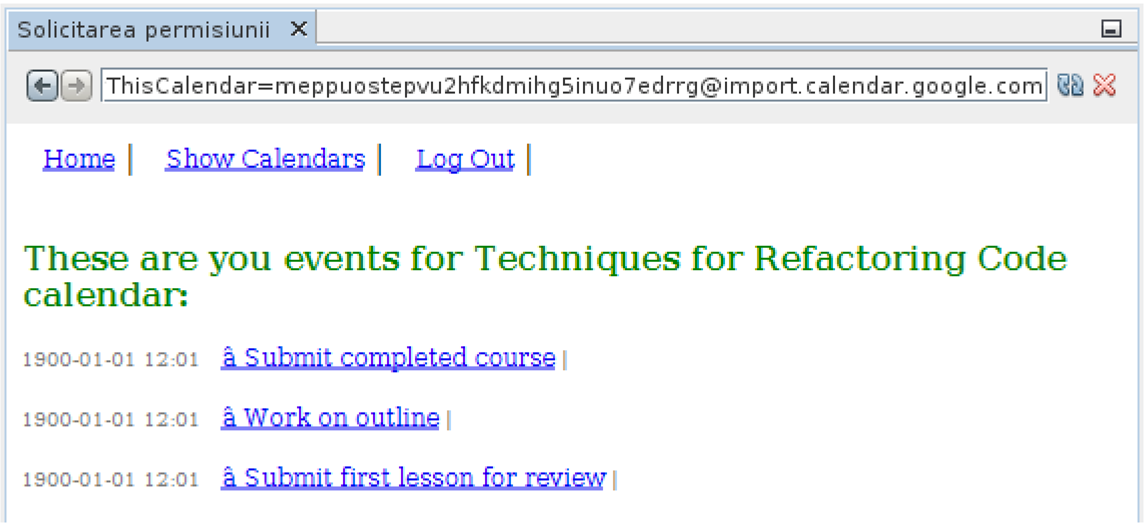


We have a single function called `doUserAction()`, which decides with a long `if-else` statement, which other methods to call based on the parameters in the `GET` variable. The methods then connect to Google calendar using the API and print out to the screen whatever we want to request.

```
function printCalendarContents($client) {
   putTitle('These are you events for ' . getCalendar($client, $_GET['showThisCalendar'])['summary'] . ' calendar:');
   foreach (retrieveEvents($client, $_GET['showThisCalendar']) as $event) {
      print('<div style="font-size:10px;color:grey;">' . date('Y-m-d H:m', strtotime($event['created'])));
      putLink('?showThisEvent=' . htmlentities($event['id']) .
            '&calendarId=' . htmlentities($_GET['showThisCalendar']), $event['summary']);
      print('</div>');
      print('<br>');
   }
}
```

This example is probably the most complicated function in our code. It calls a helper function named`putTitle()`, which just prints out some formatted HTML for the heading. The title will contain the name for our calendar which can be obtained by

calling `getCalendar()` from `functions_google_api.php`. The returned calendar will be an array, containing a `summary` field. That is what we are after.
The `$client` variable is passed all over the place in all of our functions. It is required to connect to the Google API. We will deal with this later.
Next, we cycle over all of the events in the current calendar. This list of arrays is obtained by running the API call encapsulated in `retrieveEvents()`. For each event, we print out the date it was created and then its title.


Solicitarea permisiunii
ThisCalendar=meppuostepvu2hfkdmihg5inuo7edrrg@import.calendar.google.com
Home | Show Calendars | Log Out |

These are you events for Techniques for Refactoring Code calendar:

1900-01-01 12:01 â Submit completed course |

1900-01-01 12:01 â Work on outline |

1900-01-01 12:01 â Submit first lesson for review |

The rest of the code is similar to what we've already discussed and even easier to understand. Feel free to play around with it before continuing on to the next section.

Organizing the Procedural Code

Our current code is OK, but I think we can do better and organize it in a more appropriate way. You can find the project with the completed organized code under the name "GoogleCalProceduralOrganized" in the attached source code.

Using a Global Client Variable

The first thing that annoys me about our unorganized code, is that we are passing this `$client` variable in as an argument all over the place, several levels deep within nested functions. Procedural programming has a clever way to solve this, a global variable. Since `$client` is defined in `index.php` and in the global scope, all we have to change is how our functions use it. So instead of expecting a `$client` parameter, we can use:

```
function printCalendars() {
   global $client;
```

```
    putTitle('These are your calendars:');
    foreach (getCalendarList($client)['items'] as $calendar) {
        putLink('?showThisCalendar=' . htmlentities($calendar['id']),
$calendar['summary']);
        print('<br>');
    }
}
```

Compare the current code to the newly organized code to see the difference. Instead of passing in `$client`as a parameter, we used `global $client` in all of our functions and passed it as a parameter only to the Google API functions. Technically, even the Google API functions could have used the `$client` variable from the global scope, but I think it is better to keep the API as independent as possible.

Separating Presentation From Logic

Some functions are clearly, only for printing things on the screen, others are for deciding what to do, and some are a little bit of both. When this occurs, sometimes it's best to move these specific-purpose functions into their own file. We'll start with the functions used purely for printing things to the screen, these will be moved into a `functions_display.php` file. See them below.

```
function printHome() {
    print('Welcome to Google Calendar over NetTuts Example');
}


function printMenu() {
    putLink('?home', 'Home');
    putLink('?showCalendars', 'Show Calendars');
    putLink('?logout', 'Log Out');
    print('<br><br>');
}
```

```
function putLink($href, $text) {
   print(sprintf('<a href="%s" style="font-size:12px;margin-left:10px;">%s</a> | ', $href, $text));
}

function putTitle($text) {
   print(sprintf('<h3 style="font-size:16px;color:green;">%s</h3>', $text));
}

function putBlock($text) {
   print('<div display="block">'.$text.'</div>');
}

```

The rest of this process of separating our presentation from logic requires us to extract the presentation part from our methods. Here is how we did it with one of the methods.

```
function printEventDetails() {
   global $client;
   foreach (retrieveEvents($_GET['calendarId']) as $event)
      if ($event['id'] == $_GET['showThisEvent']) {
         putTitle('Details for event: '. $event['summary']);
         putBlock('This event has status ' . $event['status']);
         putBlock('It was created at ' .
               date('Y-m-d H:m', strtotime($event['created'])) .
               ' and last updated at ' .
               date('Y-m-d H:m', strtotime($event['updated'])) . '.');
         putBlock('For this event you have to <strong>' .
$event['summary'] . '</strong>.');
      }
}
```

Clearly we can see that whatever is inside of the `if` statement is just presentation code and the rest is business logic. Instead of one bulky function handling everything, we will break it into multiple functions:

```
function printEventDetails() {
   global $client;
   foreach (retrieveEvents($_GET['calendarId']) as $event)
      if (isCurrentEvent($event))
         putEvent($event);
}

function isCurrentEvent($event) {
   return $event['id'] == $_GET['showThisEvent'];
}
```

After the separation, the business logic is now very simple. We even extracted a small method to determine if the event is the current one. All of the presentation code is now the responsibility of a function named`putEvent($event)` which resides in the `functions_display.php` file:

```
function putEvent($event) {
   putTitle('Details for event: ' . $event['summary']);
   putBlock('This event has status ' . $event['status']);
   putBlock('It was created at ' .
         date('Y-m-d H:m', strtotime($event['created'])) .
         ' and last updated at ' .
         date('Y-m-d H:m', strtotime($event['updated'])) . '.');
```

```
  putBlock('For this event you have to <strong>' . $event['summary'] .
'</strong>.');
}
```

Even though this method only displays information, we have to keep in mind that it depends on intimate knowledge about the structure of `$event`. But, this is OK for now. As for the rest of the methods, they were separated in a similar manner.

Eliminating Long if-else Statements

The last thing that bothers me about our current code is the long if-else statement in our `doUserAction()`function, which is used to decide what to do for each action. Now, PHP is quite flexible when it comes to meta-programming (calling functions by reference). This trick allows us to correlate function names with the`$_GET` variable's values. So we can introduce a single `action` parameter in the `$_GET` variable and use the value from that as a function name.

```
function doUserAction() {
   putMenu();
   if (!isset($_GET['action'])) return;
      $_GET['action']();
}
```

Based on this approach, our menu will be generated like this:

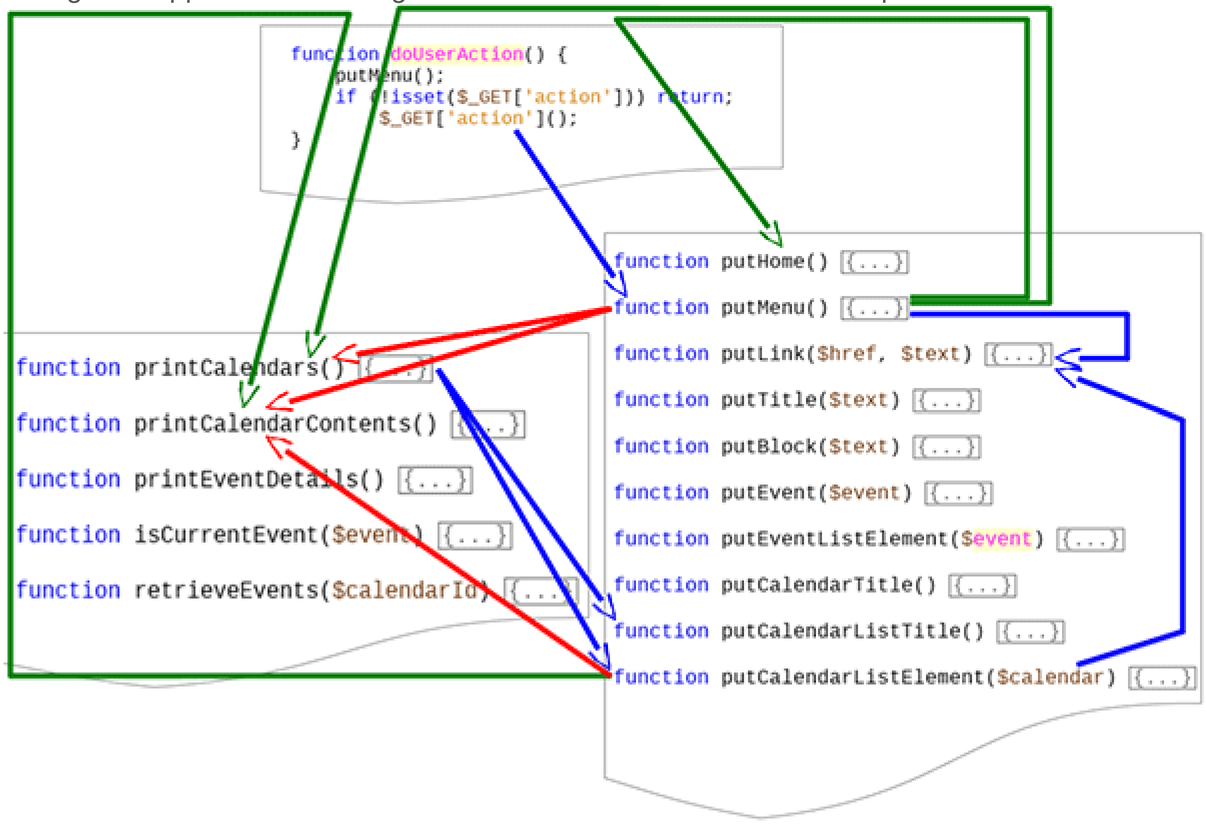
```
function putMenu() {
   putLink('?action=putHome', 'Home');
   putLink('?action=printCalendars', 'Show Calendars');
   putLink('?logout', 'Log Out');
   print('<br><br>');
}
```

As you can probably see, this reorganization already pushed us toward an object oriented design. It's not clear what kind of objects we have and with what exact behavior, but we have some clues here-and-there.

We have presentations that depend on data types from the business logic. This resembles the dependency inversion we were talking about in the introductory chapter. The flow of the control is still from the business logic towards presentation, but the source code dependency started to morph into a reversed dependency. I would say, at this point it is more like a bidirectional dependency.
Another hint of an object oriented design is the little bit of meta-programming we just did. We call a method, which we know nothing about. It can be anything and it's like we are dealing with a low level of polymorphism.

Dependency Analysis

For our current code we could draw a schema, like the one below, to illustrate the first few steps through our application. Drawing all the lines would have been too complicated.



```
function doUserAction() {
    putMenu();
    if (!isset($_GET['action'])) return;
    $_GET['action']();
}
```

```
function printCalendars() {...}
function printCalendarContents() {...}
function printEventDetails() {...}
function isCurrentEvent($event) {...}
function retrieveEvents($calendarId) {...}
```

```
function putHome() {...}
function putMenu() {...}
function putLink($href, $text) {...}
function putTitle($text) {...}
function putBlock($text) {...}
function putEvent($event) {...}
function putEventListElement($event) {...}
function putCalendarTitle() {...}
function putCalendarListTitle() {...}
function putCalendarListElement($calendar) {...}
```

We marked with blue lines, the procedure calls. As you can see they flow in the same direction as before. Additionally we have the green lines marking indirect calls. These are all passing through `doUserAction()`. These two types of lines represent the flow of control, and you can observe that it is basically unchanged.
The red lines however introduce a different concept. They represent a rudimentary source code dependency. I mean rudimentary, because it is not that obvious. The `putMenu()` method includes the names of the functions that have to be called for that particular link. This is a dependency and the same rule applies to all the other methods creating links. They *depend* on the behavior of the other functions.

A second type of dependency can also be seen here. The dependency on data. I previously mentioned`$calendar` and `$event`. The printing functions need to have intimate knowledge about the internal structure of these arrays to do their jobs.
So after all of that, I think we have plenty of reasons to move on to our last step.

An Object Oriented Solution

Regardless of the paradigm in use, there is no perfect solution for a problem. So here is how I propose to organize our code in an object oriented manner.

First Instinct

We already started to separate concerns in business logic and presentation. We even presented our`doUserAction()` method as a separate entity. So my first instinct is to create three classes `Presenter`,`Logic`, and `Router`. These will most likely change later, but we need a place to start, right?
The `Router` will contain only one method and it will remain fairly similar to the previous implementation.

```
class Router {

    function doUserAction() {
        (new Presenter())->putMenu();
        if (!isset($_GET['action']))
            return;
        (new Logic())->$_GET['action']();
    }

}
```

So now we have to explicitly call our `putMenu()` method using a new `Presenter` object and the rest of the actions will be called using a `Logic` object. However, this immediately causes a problem. We have an action that is not in the Logic class. `putHome()` is in the Presenter class. We need to introduce an action in `Logic`that will delegate to the

Presenter’s `putHome()` method. Remember, for the time being we only want to wrap our existing code into the three classes we identified as possible candidates for an OO design. We want to do only what is absolutely necessary to make the design work. After we have working code, we will change it further.

As soon as we put a `putHome()` method into the Logic class we have a dilemma. How to call methods from Presenter? Well, we could create and pass a Presenter object into Logic so it always has a reference to presentation. Let’s do that from our Router.

```
class Router {

   function doUserAction() {
      (new Presenter())->putMenu();
      if (!isset($_GET['action']))
         return;
      (new Logic(new Presenter))->$_GET['action']();
   }

}
```

Now we can add a constructor in Logic and add in the delegation toward `putHome()` in Presenter.

```
class Logic {

   private $presenter;

   function __construct(Presenter $presenter) {
      $this->presenter = $presenter;
   }
```

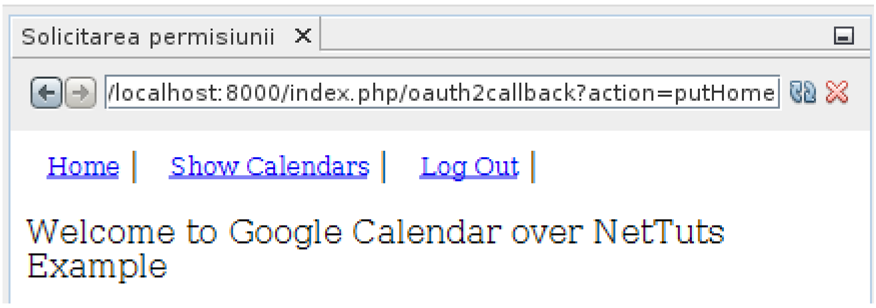
```

    function putHome() {

        $this->presenter->putHome();

    }


[...]


}

```

With a few minor adjustments in `index.php` and having Presenter wrapping the old display methods, Logic wrapping the old business logic functions, and Router wrapping the old action selector, we can actually run our code and have the “Home” menu element working.

```
require_once './google-api-php-client/src/Google_Client.php';
require_once './google-api-php-client/src/contrib/Google_CalendarService.php';
require_once __DIR__ . '/../apiAccess.php';
require_once './functins_google_api.php';
require_once './Presenter.php';
require_once './Logic.php';
require_once './Router.php';
session_start();


$client = createClient();
if(!authenticate($client)) return;

```

```php
(new Router())->doUserAction();
```

And here it is in action.


Solicitarea permisiunii
/localhost:8000/index.php/oauth2callback?action=putHome
Home | Show Calendars | Log Out |

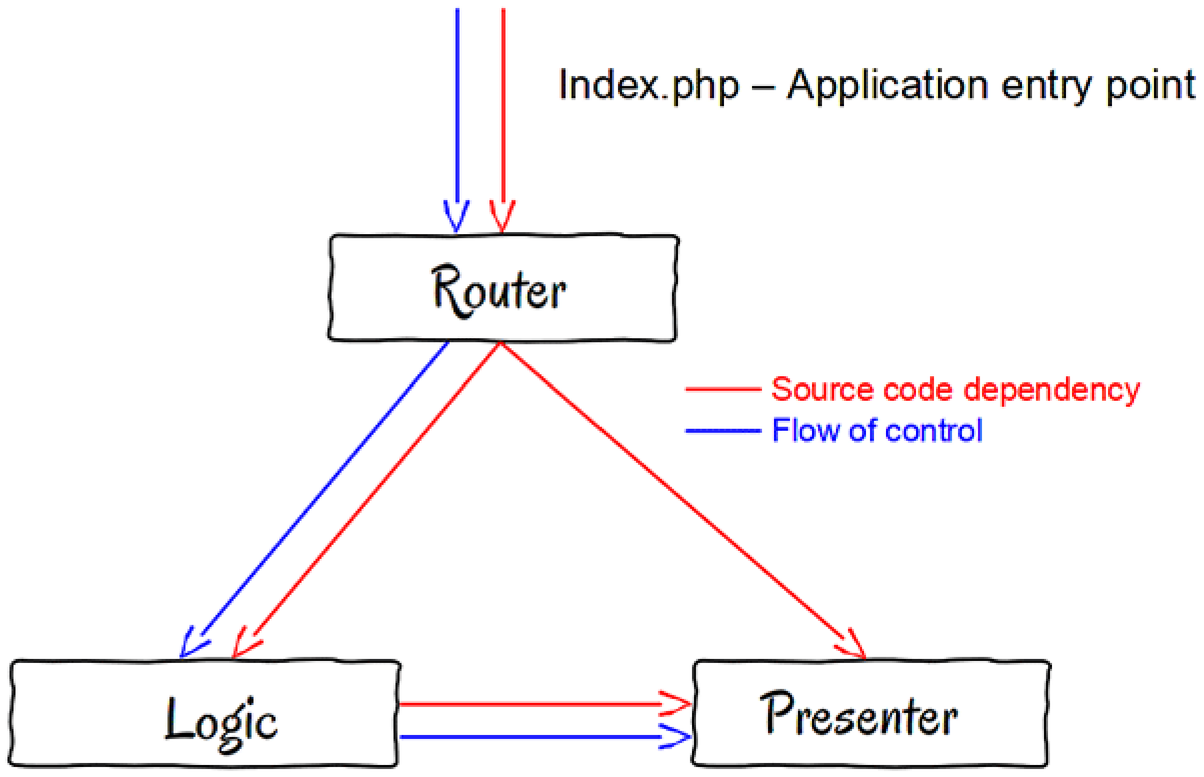
Welcome to Google Calendar over NetTuts Example

Next, in our Logic class, we need to properly change calls to display logic, to work with `$this->presenter`. Then we have two methods – `isCurrentEvent()` and `retrieveEvents()` – that are used only inside the Logic class. We will make them private and change the calls accordingly.

We will then take the same approach with Presenter class. We will change all calls to methods to point to`$this->something` and make `putTitle()`, `putLink()`, and `putBlock()` private, since they are used only from Presenter. Check out the code in the **GoogleCalObjectOrientedInitial** directory in the attached source code if you have a hard time doing all of these changes by yourself.

At this point we have a working app. It is mostly procedural code wrapped in OO syntax, which still uses the`$client` global variable and has tons of other anti-object-oriented smells, but it works.

If we draw the class diagram with dependencies for this code, it will look like this: >



Both the flow control and source code dependencies go through the router, then the logic, and finally through the presentation. This last change we did actually fades a little bit of the dependency inversion we observed in our previous step. But don't let yourself be fooled. The principle is there, we just have to make it obvious.

Reverting the Source Code Dependency

It's hard to say that one SOLID principle is more important than another, but I think the Dependency Inversion Principle has the greatest, immediate impact on your design. This principle states:

***A**: High-level modules should not depend on low-level modules. Both should depend on abstractions and **B**: Abstractions should not depend upon details. Details should depend upon abstractions.*

To put it simply, this means that concrete implementations should depend on abstract classes. As your classes become abstract, the less they tend to change. So you can perceive the problem as: frequently changing classes should depend on other, much more stable classes. So the most volatile part of any application is probably its user interface, which would be the Presenter class in our application. Let's make this dependency inversion obvious.

First we will make our Router use only the Presenter and break its dependency on Logic.

```
class Router {

    function doUserAction() {
        (new Presenter())->putMenu();
        if (!isset($_GET['action']))
            return;
        (new Presenter())->$_GET['action']();
    }

}
```

Then we will change Presenter to use an instance of Logic and ask it for the information it needs to present. In our case, I consider it acceptable for Presenter to actually create the instance of Logic, but in any production system, you will likely have Factories creating the business logic related objects and injecting them into the presentation layer.

Now, the function `putHome()`, present in both the Logic and Presenter classes, will disappear from Logic. This is a good sign, as we're removing duplication. The constructor and reference to Presenter also disappears from Logic. On the other hand, a constructor creating a Logic object has to be written on Presenter.

```
class Presenter {

    private $businessLogic;

    function __construct() {
        $this->businessLogic = new Logic();
    }

    function putHome() {
        print('Welcome to Google Calendar over NetTuts Example');

    }

[...]

}
```

After those changes, clicking on **Show Calendars** will however produce a nice error. Because all of our actions from within the links are pointing to function names in the Logic class, we will have to do some more consistent changes to reverse the dependency between the two. Let's take it one method at a time. The first error message says:

```
Fatal error: Call to undefined method Presenter::printCalendars()
in /[...]/GoogleCalObjectOrientedFinal/Router.php on line 9
```

So, our Router wants to call a method that does not exist on Presenter, `printCalendars()`. Let's create that method in Presenter and check what it did in Logic. It printed a title and then cycled through some calendars and called `putCalendar()`. In Presenter the `printCalendars()` method will look like this:

```
function printCalendars() {
    $this->putCalendarListTitle();
    foreach ($this->businessLogic->getCalendars() as $calendar) {
        $this->putCalendarListElement($calendar);
    }
}
```

On the other hand, in Logic, the method becomes quite anemic. Just a forward call to the Google API library.

```
function getCalendars() {
   global $client;
   return getCalendarList($client)['items'];
}
```

This may make you ask yourself two questions, “Do we actually have a need for a Logic class?” and “Does our application even have any logic?”. Well, we don’t know yet. For the time being we will continue the process above, until all of the code works, and Logic does not depend on Presenter anymore.

So, we will use a `printCalendarContents()` method in Presenter, like the one below:

```
function printCalendarContents() {
   $this->putCalendarTitle();
   foreach ($this->businessLogic->getEventsForCalendar() as $event) {
      $this->putEventListElement($event);
   }
}
```

Which in turn, will allow us to simplify the `getEventsForCalendar()` in Logic, into something like this.

```
function getEventsForCalendar() {
   global $client;
   return getEventList($client, htmlspecialchars($_GET['showThisCalendar']))['items'
}
```

Now this works, but I have a concern here. The `$_GET` variable is being used in both the Logic and Presenter classes. Shouldn’t only the Presenter class be using `$_GET`? I mean, Presenter absolutely needs to know about `$_GET` because it has to create links which are populating this `$_GET` variable. So that would mean that `$_GET` is strictly HTTP related. Now, we want our code to work with a CLI or desktop graphical UI. So we want to keep this knowledge in only the Presenter. This makes the two methods from above, transform into the two below.

```php
function getEventsForCalendar($calendarId) {
   global $client;
   return getEventList($client, $calendarId)['items'];
}
```

```php
function printCalendarContents() {
   $this->putCalendarTitle();
   $eventsForCalendar = $this->businessLogic->getEventsForCalendar(htmlspecialchars($_GET['showThisCalendar']));
   foreach ($eventsForCalendar as $event) {
      $this->putEventListElement($event);
   }
}
```

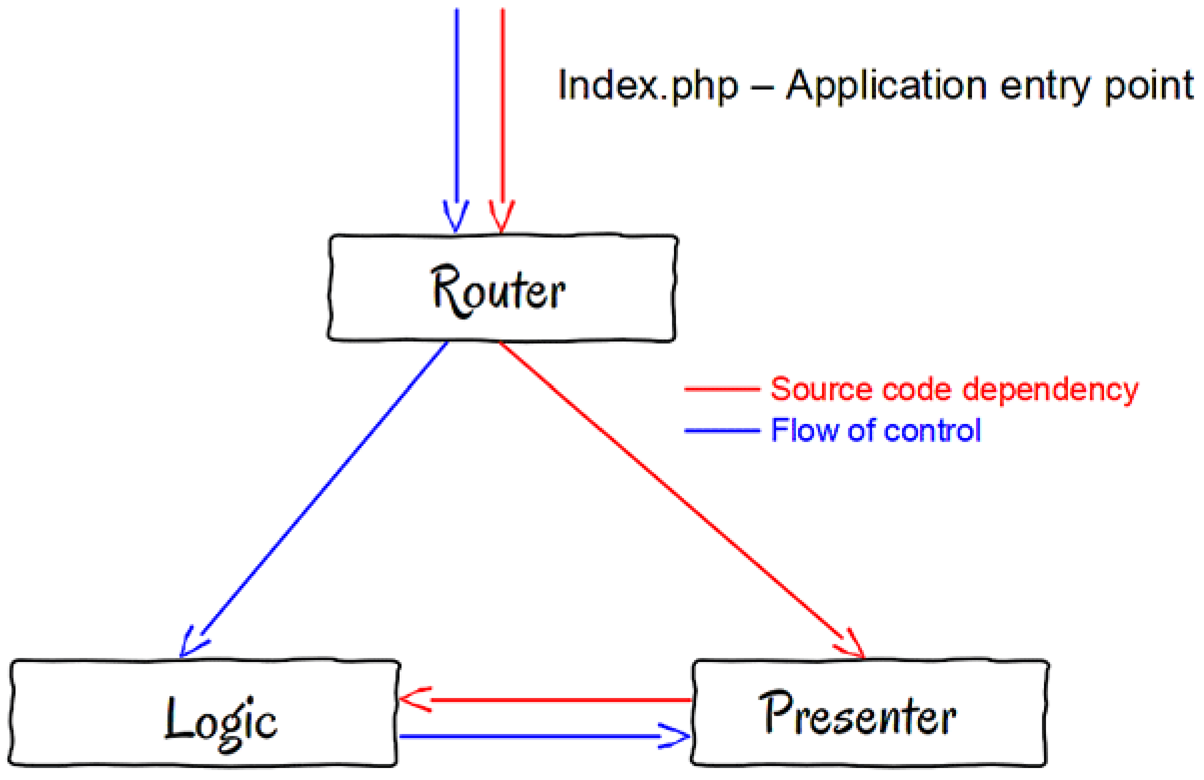
Now the last function we have to deal with is for printing a specific event. For the sake of this example, suppose there is no way we can retrieve an event directly and we have to find it by ourselves. Now our Logic class comes in handy. It is a perfect place to manipulate lists of events and search for a specific ID:

```php
function getEventById($eventId, $calendarId) {
   foreach ($this->getEventsForCalendar($calendarId) as $event)
      if ($event['id'] == $eventId)
         return $event;
}
```

And then the corresponding call on Presenter will take care of printing it:

```
function printEventDetails() {
   $this->putEvent(
      $this->businessLogic->getEventById(
         $_GET['showThisEvent'],
         $_GET['calendarId']
      )
   );
}
```

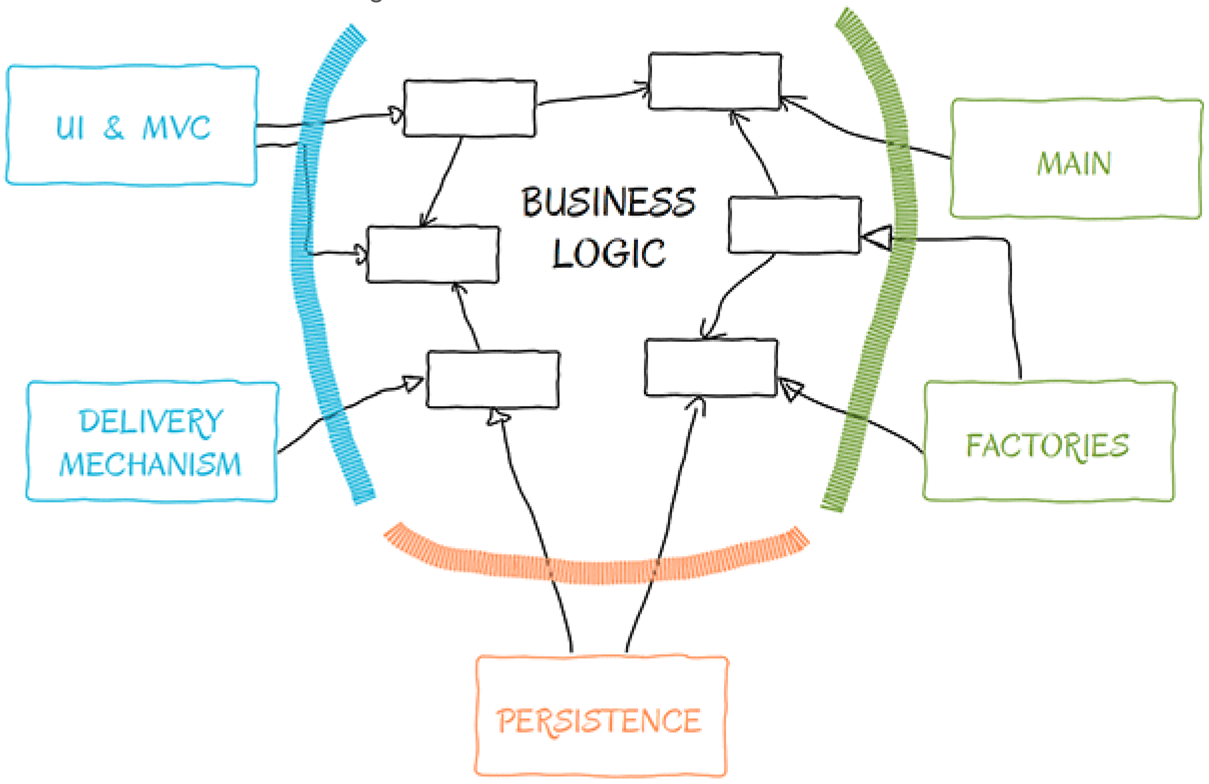
That’s it. Here we are. Dependency inverted!



Control still flows from Logic towards Presenter. The content presented is totally defined by Logic. If, for example tomorrow, we want to connect to another calendar service, we can create another Logic, inject it into Presenter, and Presenter will not even notice any difference. Also, the source code dependency was inverted successfully. Presenter is the only one that creates and directly depends on Logic. This dependency is crucial in allowing Presenter to change how it shows data, without effecting anything in Logic. Additionally, it allows us to switch our HTML Presenter with a CLI Presenter or any other method of displaying the information to the user.

Getting Rid of the Global Variable

Probably the last remaining serious design flaw is the use of a global variable for `$client`. All of the code in our application has access to it. Miraculously, the only class actually needing `$client` is our Logic class. The obvious step is to make a private class variable. But doing so requires us to propagate `$client`through the Router, into the Presenter, so that it can create a Logic object with the `$client` variable. This solves little of our problems. We need to build our classes in an isolated place and properly inject the dependencies into each other. For any larger class structure we would use Factories, but for our small example, the `index.php` file will be a great place to hold the creation logic. And being the entry point to our application, aka the "main" file in the high level architecture schema, it is still outside the boundaries of our business logic.



So we change `index.php` into the code below, keeping all of the includes and the session_start() command:

```
$client = createClient();
if(!authenticate($client)) return;

```

```
$logic = new Logic($client);

$presenter = new Presenter($logic);

(new Router($presenter))->doUserAction();
```

Final Thoughts

And we’re finished. There are surely some other things we could do to make our design even better. If nothing else, we could write a couple tests for our methods on the Logic class. Maybe our Logic class could also be renamed to something more representative, like GoogleCalendarGateway. Or we could create Event and Calendar classes to even better control the data and behavior on these concepts and break the Presenter’s dependency on an array for these data types. Another improvement and extension would be to actually create polymorphic action classes instead of just calling functions by reference from `$_GET`. There are endless little things we could do to further improve even this simple design. I will let you have this great opportunity to experiment, starting from this final version of the code you can find in the attached archive under the **GoogleCalObjectOrientedFinal** directory.
If you are adventurous, you can extend this little application to connect to other calendar services and present information in different ways and on different platforms. For all of you using NetBeans, each source code folder contains a NetBeans project, so you should be able to directly open them. In the final version, PHPUnit is also prepared for you, but in rest of the projects, I removed it because we did no testing.

Thank you for reading.