Agile Toolkit Core

Release 0.1.0

May 05, 2017

Contents

Contents:

CHAPTER 1

Overview

Agile Core is a collection of PHP Traits for designing object-oriented frameworks

Run-Time Tree (containers)

When you want your framework to look after relationships between objects by implementing containers, you can use *ContainerTrait*.

You will be able to use `ContainerTrait::getElement()` to access elements inside container:

```
$object->add(new AnoterObject(), 'test');
$another_object = $object->getElement('test');
```

If you additionally use *TrackableTrait* then your objects also receive unique "name". From example above:

- $object->name == "app_object_4"
- $another_object->name == "app_object_4_test"

Initializers

When object is created, the constructor is executed. Sometimes your object needs to be aware of it's environment, and that's why initializer will allow your developers to extend init() method, that will be called after your new object is integrated into the environment, if you use *InitializerTrait*:

```
class MyClass {
    use InitializerTrait;

    function init()
    {
        parent::init();
```

```
        // some variables are available here
        // echo $this->owner;
        // echo $this->name;
        // echo $this->app;
    }
}
```

Factory

Normally you can only add existing objects into your run-time tree. *FactoryTrait* will allow you to specify the class name:

```
$object->add('OtherObject');
```

This will also enable similar features for *Modelable objects*:

```
$object->setModel('MyModel');
// same as
$object->setModel(new Model_MyModel);
```

You can specify namespaces with backslash or regular slash:

```
$object->setModel('data/MyModel');
// same as
$object->setModel(new data\Model_MyModel);
```

Dynamic Methods

DynamicMethodTrait adds ability to add methods into objects dynamically. That's like a "trait" feature of a PHP, but implemented in run-time:

```
$object->addMethod('test', function($o, $args){ echo 'hello, '.$args[0]; } );
$object->test('world');
// outputs: hello, world
```

There are also methods for removing and checking if methods exists, so:

```
method_exists($object, 'test');
// now should use
$object->hasMethod('test');

// and this way you can remove method
$object->removeMethod('test');
```

Hooks

HookTrait adds and trigger hooks for objects:

```
$object->addHook('test', function($o){ echo 'hello'; }
$object->addHook('test', function($o){ echo 'world'; }

$object->hook('test');
// outputs: helloworld
```

Modelable Objects

In an MVP concept you have 3 types of objects - Models, Views and Presenter. The Presenter is responsible for creating and linking View and Model together.

Views are generic presentation widgets that can gain some insight into your data through the Model declaration.

`ModelableTrait` allows you to associate object with a Model:

```
$form->setModel('Order');

// or

$grid->setModel($order->ref('Items'), ['name', 'qty', 'price']);
```

Quick Exception

When you are throwing exception somewhere in your logic, you have to collect enough information about the context. Sometimes it's easier to let your framework do it for you. Add `QuickExceptionTrait` and you can throw exceptions like this:

```
throw $object->exception(['Incorrect foo value', 'foo'=>$bar]);
```

This is similar to the regular exception, however in addition to back-trace this will capture information about $object. This object will also be able to add more information into your query:

```
throw $db->exception('Bad Query', 'QueryException');

class QueryException extends Exception {
    protected $query;

    function __construct($object){
        $this->query = $object->getDebugQuery();
    }
}
```

App Scope

Typical software design will create the application scope. Most frameworks relies on "static" properties, methods and classes. This puts some limitations on your implementation (you can't have multiple applications).

AppScopeTrait will pass the 'app' property into all objects that you're adding, so that you know for sure which application you work with:

```
$object1->add('Object2');

class Object2 {
    use AppScopeTrait;
    use InitializerTrait;

    function init() {

        parent::init();

        echo 'app is = '.$this->app;
    }
}
```

Session

When application is executed in environment, some objects of the applications may want to "record their state" in session scope. Technically this could be routed through the data source in the application that handles the session, but PHP has a wonderful support for $_SESSION already.

`SessionTrait` makes it possible for objects to have unique data-store inside a session.

This feature can be used by Views / Widgets that needs session info.

Syntax:

```
$this->setField('search', $this->recall('search', null));

// on submit

$this->memorize('search', $_POST['search']);
```

The session store is unique for each object identified by their "name" property.

DebugTrait

`DebugTrait` allows your objects to execute:

```
$object->debug();
$object->log('something happened');
$object->warn('bad things happen');
```

The debug will only be collected if the debug mode is turned on, otherwise calls to log() and warn() will be ignored.

CHAPTER 2

Exception

class Exception

Introduction

Exception provides several improvements over vanilla PHP exception class. The most significant change is introduction of parameters.

property Exception::$**params**

Parameters will store supplimentary information that can help identify and resolve the problem. There are two ways to supply params, either during the constructor or using addMoreInfo()

Exception::**__construct**(*error*, *code*, *previous*)

This uses same format as a regular PHP exception, but error parametetr will now support array:

```
throw new Exception(['Value is too big', 'max'=>$max]);
```

The other option is to supply error is:

Exception::**addMoreInfo**(*param*, *value*)

Augments exception by providing extra information. This is a typical use format:

```
try {
    $field->validate();
} catch (Validation_Exception $e) {
    $e->addMoreInfo('field', $field);
    throw $e;
}
```

I must note that the reason for using parameters is so that the name of the actual exception could be localized easily.

The final step is to actually get all the information from your exception. Since the exception is backwards compatible, it will contain message, code and previous exception as any normal PHP excetpion would, but to get the parameters you would need to use:

`Exception::getParams()`
Return array that lists all params collected by exception.

Some param values may be objects.

`Exception::setMessage(`*$message*`)`
Change message (subject) of a current exception. Primary use is for

localization purposes.

Output Formatting

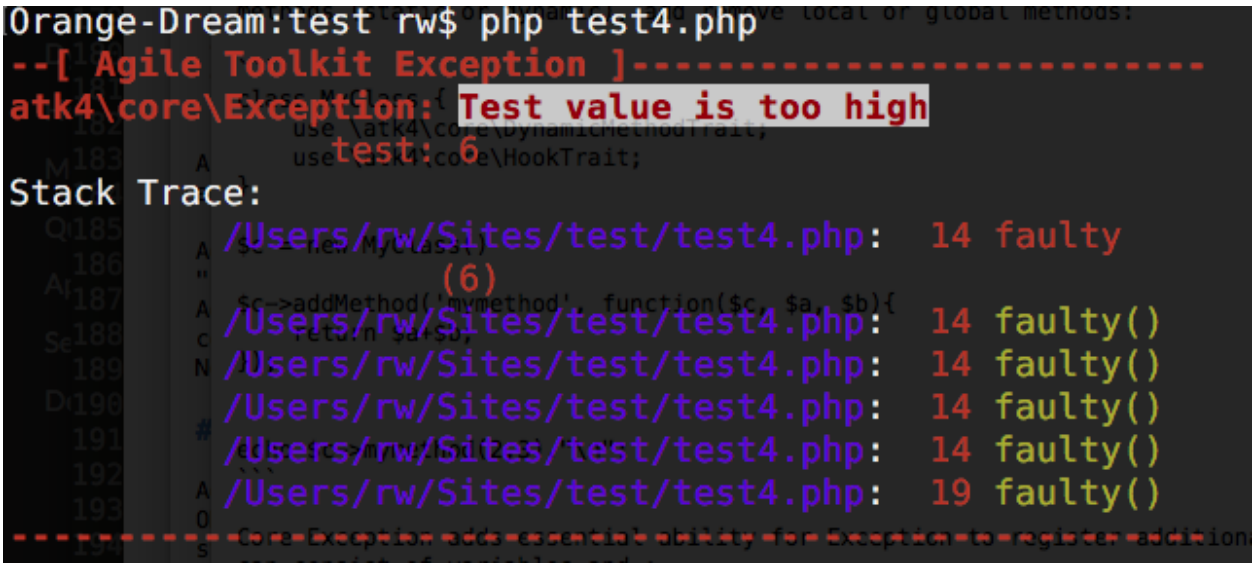
Exception (at least for now) contains some code to make the exception actually look good. This functionality may be removed in the later versions to to facilitate use of proper loggers. For now:

`Exception::getColorfulText()`

Will return nice ANSI-colored exception that you can output to the console for user to see. This will include the error, params and backtrace. The code will also make an attempt to locate and highlight the code that have caused the problem.

`Exception::getColorfulText()`

Will return nice HTML-formatted exception that will rely on a presence of Semantic UI. This will include the error, params and backtrace. The code will also make an attempt to locate and highlight the code that have caused the problem.



CHAPTER 3

Run-Time Tree (Containers)

There are two relevant traits in the Container mechanics. Your "container" object should implement *ContainerTrait* and your child objects should implement *TrackableTrait* (if not, the $owner/$elements links will not be established)

If both parent and child implement *AppScopeTrait* then the property of `AppScopeTrait::app` will be copied from parent to the child also.

If your child implements *InitializerTrait* then the method *InitializerTrait::init* will also be invoked after linking is done.

You will be able to use `ContainerTrait::getElement()` to access elements inside container:

```
$object->add(new AnoterObject(), 'test');
$another_object = $object->getElement('test');
```

If you additionally use *TrackableTrait* then your objects also receive unique "name". From example above:

- $object->name == "app_object_4"
- $another_object->name == "app_object_4_test"

Name Trait

trait `ObjectTrait`
Name trait only adds the 'name' property. Normally you don't have to use it because *TrackableTrait* automatically inherits this trait. Due to issues with PHP5 if both *ContainerTrait* and *TrackableTrait* are using `NameTrait` and then both applied on the object, the clash results in "strict warning". To avoid this, apply `NameTrait` on Containers only if you are NOT using *TrackableTrait*.

Properties

property `ObjectTrait::$`**`name`**
Name of the object.

Methods

None

Container Trait

trait `ContainerTrait`

If you want your framework to keep track of relationships between objects by implementing containers, you can use *`ContainerTrait`*. Example:

```
class MyContainer extends OtherClass {
    use atk4\core\ContainerTrait;

    function add($obq, $args = []) {
        return $this->_add_Container($obj, $args);
    }
}

class MyItem  {
    use atk4\core\TrackableTrait;
}

Now the instances of MyItem can be added to instances of MyContainer and can keep␣
↪track::

$parent = new MyContainer();
$parent->name = 'foo';
$parent->add(new MyItem(), 'child1');
$parent->add(new MyItem());

echo $parent->getElement('child1')->name;
// foo_child1

if ($parent->hasElement('child1')) {
    $parent->removeElement('child1');
}

$parent->each(function($child) {
    $child->doSomething();
});
```

Child object names will be derived from the parent name.

Properties

property `ContainerTrait::$elements`

Contains a list of objects that have been "added" into the current container. The key is a "shot_name" of the child. The actual link to the element will be only present if child uses trait "TrackableTrait", otherwise the value of array key will be "true".

Methods

Trackable Trait

trait `TrackableTrait`

Trackable trait implements a few fields for the object that will maintain it's relationship with the owner (parent).

When name is set for container, then all children will derive their names of the parent.

- Parent: foo
- Child: foo_child1

The name will be unique within this container.

Properties

property `TrackableTrait::$owner`

Will point to object which has add()ed this object. If multiple objects have added this object, then this will point to the most recent one.

property `TrackableTrait::$short_name`

When you add item into the owner, the "short_name" will contain short name of this item.

Methods

CHAPTER 4

Initializer Trait

trait `InitializerTrait`

Introduction

With our traits objects now become linked with the "owner" and the "app". Initializer trait allows you to define a method that would be called after object is linked up into the environment.

Declare a object class in your framework:

```
class FormField {
    use AppScopeTrait;
    use TrackableTrait;
    use InitializerTrait;

}

class FormField_Input extends FormField {

    public $value = null;

    function init() {
        parent::init();

        if($_POST[$this->name) {
            $this->value = $_POST[$this->name];
        }
    }

    function render() {
        return '<input name="'.$this->name.'" value="'.$value.'"/>';
    }
}
```

Properties

property `InitializerTrait::$_initialized`
Internal property to make sure you have called parent::init() properly.

Methods

`InitializerTrait::init()`
A blank init method that should be called. This will detect the problems when init() methods of some of your base classes has not been executed and prevents from some serious mistakes.

If you wish to use traits class and extend it, you can use this in your base class:

```
class FormField {  use AppScopeTrait; use TrackableTrait; use InitializerTrait {
        init as _init
    }
    public $value = null;
    function init() {  $this->_init(); // call init of InitializerTrait
        if($_POST[$this->name) {  $this->value = $_POST[$this->name];
        }
    }
}
```

CHAPTER 5

Factory Trait

trait `FactoryTrait`

Introduction

This trait can be used to dynamically create new objects by their class names. It also contains method for class name normalization.

Properties

None

Methods

CHAPTER 6

Dynamic Method Trait

trait `DynamicMethodTrait`

Introduction

Adds ability to add methods into objects dynamically. That's like a "trait" feature of a PHP, but implemented in run-time:

```
$object->addMethod('test', function($o, $args){ echo 'hello, '.$args[0]; } );
$object->test('world');
```

Global Methods

If object has application scope *`AppScopeTrait`* and the application implements *`HookTrait`* then executing $object->test() will also look for globally-registered method inside the application:

```
$object->app->addGlobalMethod('test', function($app, $o, $args){
    echo 'hello, '.$args[0];
});

$object->test('world');
```

Of course calling test() on the other object afterwards will trigger same global method.

If you attempt to register same method multiple times you will receive an exception.

Dynamic Method Arguments

When calling dynamic method first argument which is passed to the method will be object itself. Dynamic method will also receive all arguments which are given when you call this dynamic method:

```
$m->addMethod('sum', function($m, $a, $b){ return $a+$b; });
echo $m->sum(3,5);
// 8
```

Properties

None

Methods

DynamicMethodTrait::**tryCall**(*$method*, *$arguments*)
: Tries to call dynamic method, but doesn't throw exception if it is not possible.

DynamicMethodTrait::**addMethod**(*$name*, *$callable*)
: Add new method for this object. See examples above.

DynamicMethodTrait::**hasMethod**(*$name*)
: Returns true if object has specified method (either native or dynamic). Returns true also if specified methods is defined globally.

DynamicMethodTrait::**removeMethod**(*$name*)
: Remove dynamically registered method.

DynamicMethodTrait::**addGlobalMethod**(*$name*, *$callable*)
: Registers a globally-recognized method for all objects.

DynamicMethodTrait::**hasGlobalMethod**(*$name*)
: Return true if such global method exists.

CHAPTER 7

Hook Trait

trait `HookTrait`

Introduction

HookTrait adds some methods into your class to registering call-backs that would be executed by triggering a hook. All hooks are local to the object, so if you want to have application-wide hook then use *app* property.

Hook Spots

Hook is described by a string identifier which we call hook-spot, which would normally be expressing desired action with prefixes “before” or “after if necessary.

Some good examples for hook spots are:

- beforeSave
- afterDelete
- validation

The framework or application would typically execute hooks like this:

```
$obj->hook('spot');
```

You can register multiple call-backs to be executed for the requested *spot*:

```
$obj->addHook('spot', function($obj){ echo "Hook 'spot' is called!"; });
```

Adding callbacks

HookTrait::**addHook**(*$spot*, *$callback*, *$args = null*, *$priority = 5*)

Register a call-back method. Calling several times will register multiple callbacks which will be execute in the order that they were added.

Short way to describe callback method

There is a consise syntax for using $callback by specifying object only. In this case a metod with same name as $spot will be used as callback:

```
function init() {
    parent::init();

    $this->addHook('beforeUpdate', $this);
}

function beforeUpdate($obj){
    // will be called from the hook
}
```

Callback execution order

$priority will make hooks execute faster. Default priority is 5, but if you add hook with priority 1 it will always be executed before any hooks with priority 2, 3, 5 etc.

Normally hooks are executed in the same order as they are added, however if you use negative priority, then hooks will be executed in reverse order:

```
$obj->addHook('spot', third,    null, -1);

$obj->addHook('spot', second,   null, -5);
$obj->addHook('spot', first,    null, -5);

$obj->addHook('spot', fourth,   null, 0);
$obj->addHook('spot', fifth,    null, 0);

$obj->addHook('spot', ten,      null, 1000);

$obj->addHook('spot', sixth,    null, 2);
$obj->addHook('spot', seventh,  null, 5);
$obj->addHook('spot', eight);
$obj->addHook('spot', nine,     null, 5);
```

HookTrait::**hook**(*$spot*, *$args = null*)

execute all hooks in order. Hooks can also return some values and those values will be placed in array and returned by hook():

```
$mul = function($obj, $a, $b) {
    return $a*$b;
};
```

```
$add = function($obj, $a, $b) {
    return $a+$b;
};

$obj->addHook('test', $mul);
$obj->addHook('test', $add);

$res1 = $obj->hook('test', [2, 2]);
// res1 = [4, 4]

$res2 = $obj->hook('test', [3, 3]);
// res2 = [9, 6]
```

Arguments

As you see in the code above, we were able to pass some arguments into those hooks. There are actually 3 sources that are considered for the arguments:

- first argument to callbacks is always the $object
- arguments passed as 3rd argument to addHook() are included
- arguments passed as 2nd argument to hook() are included

You can also use key declarations if you wish to override arguments:

```
// continue from above example

$pow = function($obj, $a, $b, $power) {
    return pow($a, $power)+$pow($b, $power);
}

$obj->addHook('test', $pow, [2]);
$obj->addHook('test', $pow, [7]);

// execute all 3 hooks
$res3 = $obj->hook('test', [2, 2]);
// res3 = [4, 4, 8, 256]

$res4 = $obj->hook('test', [2, 3]);
// res3 = [6, 5, 13, 2315]
```

Breaking Hooks

`HookTrait::breakHook()`

When this method is called from a call-back then it will cause all other callbacks to be skipped.

If you pass $return argument then instead of returning all callback retutrn values in array the $return will be returned by hook() method.

If you do not pass $return value (or specify null) then list of the values collected so far will be returned

Remember that adding breaking hook with a lower priority can prevent other call-backs from being executed:

```
$obj->addHook('test', function($obj){
    $obj->breakHook("break1");
});

$obj->addHook('test', function($obj){
    $obj->breakHook("break2");
}, null, -5);

$res3 = $obj->hook('test', [4, 4]);
// res3 = "break2"
```

breakHook method is implemented by throwing a special exception that is then caught inside hook() method.

Using references in hooks

In some cases you want hook to change certain value. For example when model value is set it may call normalization hook (methods will change $value):

```
function set($field, $value) {
    $this->hook('normalize', [&$value]);
    $this->data[$field] = $value;
}

$m->addHook('normalize', function(&$a) { $a = trim($a); });
```

Checking if hook has callbacks

`HookTrait::hookHasCallbacks()`

This method will return true if at least one callback has been set for the hook.

CHAPTER 8

AppScope Trait

trait `AppScopeTrait`

Introduction

Typical software design will create the application scope. Most frameworks relies on "static" properties, methods and classes. This does puts some limitations on your implementation (you can't have multiple applications).

App Scope will pass the 'app' property into all the object that you're adding, so that you know for sure which application you work with:

Properties

property `AppScopeTrait::$app`
: Always points to current Application object

property `AppScopeTrait::$max_name_length`
: When using mechanism for ContainerTrait, they inherit name of the parent to generate unique name for a child. In a framework it makes sense if you have a unique identifiers for all the objects because this enables you to use them as session keys, get arguments, etc.

  Unfortunatelly if those keys become too long it may be a problem, so ContainerTrait contains a mechanism for auto-shortening the name based around max_name_length. The mechanism does only work if AppScopeTrait is used, $app property is set and has a max_name_length defined. Minimum value is 20.

property `AppScopeTrait::$unique_hashes`
: As more names are shortened, the substituted part is being placed into this hash and the value contains the new key. This helps to avoid creating many sequential prefixes for the same character sequenece.

Methods

None

CHAPTER 9

Dependency Injection Containers

Agile Core implements basic support for Dependency Injection Container.

What is Dependency Injection

As it turns out many PHP projects have built objects which hard-code dependencies on another object/class. For instance:

```
$book = new Book();
$book->name = 'foo';
$book->save();          // saves somewhere??
```

The above code uses some ORM notation and the book record is saved into the database. But how does Book object know about the database? Some frameworks thought it could be a good idea to use GLOBALS or STATIC. PHP Community is fighting against those patterns by using Dependency Injection which is a pretty hot topic in the community.

In Agile Toolkit this has never been a problem, because all of our objects are designed without hard dependencies, globals or statics in the first place.

"Dependency Injection" is just a fancy word for ability to specify other objects into class constructor / property:

```
$book = new Book($mydb);
$book['name'] = 'foo';
$book->save(); // saves to $mydb
```

What is Dependency Injection Container

By design your objects should depend on as little other objects as possible. This improves testability of objects, for instance. Typically constructor can be good for 1 or 2 arguments.

However in Agile UI there are components that are designed specifically to encapsulate many various objects. CRUD for example is a fully-functioning editing solution, but suppose you want to use custom form object:

```
$crud = new CRUD([
    'formEdit' => new MyForm(),
    'formAdd'  => new MyForm()
]);
```

In this scenario you can't pass all of the properties to the constructor, and it's easier to pass it through array of key/values. This pattern is called Dependency Injection Container. Theory states that developers who use IDEs extensively would prefer to pass "object" and not "array", however we typically offer a better option:

```
$crud = new CRUD();
$crud->formEdit = new MyForm();
$crud->formAdd  = new MyForm();
```

How to use DIContainerTrait

Calling this method will set object's properties. If any specified property is undefined then it will be skipped. Here is how you should use trait:

```
class MyObj {
    use DIContainerTrait;

    function __construct($defaults = []) {
        $this->setDefaults($defaults, true);
    }
}
```

You can also extend and define what should be done if non-property is passed. For example Button component allows you to pass value of $content and $class like this:

```
$button = new Button(['My Button Label', 'red']);
```

This is done by overriding setMissingProperty method:

```
class MyObj {
    use DIContainerTrait {
        setMissingProperty as _setMissingProperty;
    }

    function __construct($defaults = []) {
        $this->setDefaults($defaults, true);
    }

    function setMissingProperty($key, $value, $strict = false) {
        // do something with $key / $value

        // will either cause exception or will ignorance
        $this->_setMissingProperty($key, $value, $strict);
    }
}
```

Index