JavaScript Method Modification

Aspect Oriented Function Composition

webtechconf - Munich, October 30th 2013 - **Peter Seliger** / **@petsel**

Frontend Engineer at **XING AG**

Agenda

- Method Modification/Modifiers - Why?
- Aspect Oriented Programming (AOP) in JavaScript - Why?
- Basic Method Modifiers vs True Aspect Oriented Systems.
- Questions / Live Demo
- Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature.
- `Joinpoint`, `Pointcut`, `Advice` and `Aspect` from a JavaScript Point of View.
- Additional Features that could be provided.
- API of an already operating AO System (sneak preview).

Method Modification/Modifiers

Method Modification/Modifiers - Why?

- There are cases where one does not own the code of a method that's functionality has to be modified ...
- e.g. enriching an existing implementation with additional behavior and influencing the control flow of this new ***function composition***.
- This is the point one is in need of a set of basic AOP inspired method modifiers like
  - `Function.prototype[before|after|around]`
- Also if one uses **Function Based Object/Type Composition** patterns like **`Traits` and `Mixins`** ...
- with implementations that introduce competing methods,
- ***resolving conflicts*** can be handled easier by such modifiers.

Method Modification/Modifiers - Why?

example - enriching an existing implementation with additional behavior and influencing its control flow.

```
var requestBasketUpdate = function (evt, onSuccess) {

  var
    control   = evt.target,
    form      = control.form
  ;
  showMiniBasket();

  LightboxController.switchOnLoadingState();
  $.ajax({

    cache    : false,
    url      : form.action,
    type     : form.method.toUpperCase(),
    data     : $(form).serialize(),
    dataType : "text",

    success  : onSuccess.before(
```

Method Modification/Modifiers

Function.prototype.before

```
Function.prototype.before = function (behaviorBefore, target) {
  var proceedAfter = this;

  return function () {
    var args = arguments;

    behaviorBefore.apply(target, args);
    return proceedAfter.apply(target, args);
  };
};



var hi = function () {console.log("hi");};
var ho = function () {console.log("ho");};

hi(); // "hi"
ho(); // "ho"
```

Method Modification/Modifiers

Function.prototype.after

```
Function.prototype.after = function (behaviorAfter, target) {
  var proceedBefore = this;

  return function () {
    var args = arguments;

    proceedBefore.apply(target, args);
    return behaviorAfter.apply(target, args);
  };
};



var he = function () {console.log("he");};

he(); // "he"
ho(); // "ho"
```

Method Modification/Modifiers

Function.prototype.around

```
Function.prototype.around = function (behaviorAround, target) {
  var proceedEnclosed = this;

  return function () {
    return behaviorAround.call(target, proceedEnclosed,
                behaviorAround, arguments, target);
  };
};



var hehiho = hi.around(function (proceed, around, args, target) {

  he();
  proceed();
  ho();
```

Aspect Oriented Programming in JavaScript

Aspect Oriented Programming (AOP) in JavaScript - Why?

Aspect Oriented Programming (AOP) in JavaScript - Why?

Basic Modifiers vs True AO Systems

- Pure Method Modifying ...
- relies on direct access to every possible modifiable method.
- needs to be done explicitly for every identified method .
- lacks abstraction for the 2 last mentioned shortcomings.
  - Thus making modularized code reuse of additionally to be wrapped behavior not that handy.
- is an appealing approach for less complex tasks.
- should be seen as pre-stage for AO Systems that have to provide both abstraction and support for better code reuse.

Questions?

Live Demo

- open **twitter.com**
- open dom inspector/console
- make sure that popups are allowed
- throw the code that has been linked beneath onto the console
- play with the given example as adviced in the bottom most comments

- gist: **ao module dependencies** (raw)
- gist: **proof of concept - example: log twitter api activities**

- **modification.ao - JavaScript implementation of an AO System**

Aspect Oriented Programming in JavaScript

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

- runtime based only and not using any kind of JavaScript "transpilers" or JavaScript build tools for "code weaving" as in e.g. AspectJ.
- thus being forced focusing on what ES3 language core does provide.

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

- implementation of prototypal method modifiers e.g. `around`, `before`, `after`, `afterThrowing`, `afterReturning`, as kind of a minimal AOP influenced base set that already supports library (framework) agnostic modification of function based control flow by just wrapping additional behaviors (advice handlers) around existing methods(functions).
- clarify role of `Joinpoint`, `Pointcut`, `Advice` and `Aspect`; especially from this point of view of what makes them distinct from existing approaches in compiled and/or non dynamic and/or non functional programming languages.

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

example code (will be explained)

```
// [VariationsController] depended method modification
SubmitController.isAnyToSubmit = SubmitController.isAnyToSubmit
  .around(function (isAnyToSubmit, interceptor, args, target) {

    var evt = args[0];
    evt.isColorChanged = true;

    return isAnyToSubmit.call(target, evt); // proceed
  })
;

// [SubmitController] specific functionality
SubmitController.isAnyToSubmit = function (evt) {
  var isChecked =
    evt.isColorChanged
    || sizeRadioInputs.toArray().some(function (elm/*, idx, arr*/) {

      return elm.checked;
```

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

Joinpoint

- A `Joinpoint` in JavaScript always needs to feature both a method that is bound to an object and this very object itself (regardless of either this couple is locally scoped or not). One might even think about a label that optionally gets assigned to a joinpoint.
- Thus a joinpoint will be constructed at least from a method's name and this method's target object.

```
/*var jpIsAnyToSubmit = */ao.Joinpoint.add({
  target       : SubmitController
  methodName   : "isAnyToSubmit",
//label        : "controllers.SubmitController.isAnyToSubmit",
});
```

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

Pointcut

- A `Pointcut` in JavaScript always should be able to return a *collection of joinpoints* that are filtered according to certain criteria.
- Thus a pointcut explicitly will be constructed from its filter method.

```
var pcIsAnyToSubmit = ao.Pointcut.add({
//id       : "isAnyToSubmit", // if omitted UUID will be generated
  filter   : function (jp) {
    return (jp.getMethodName() == "isAnyToSubmit");

  //return (jp.getLabel().indexOf("...") >= 0);
  //return (jp.getTarget() === ...);
  }
});
```

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

Advice

- An Advice in JavaScript always should feature both a method that defines behavior (or could be seen as advice handler) and a named qualifier or type.
- Thus an advice will be constructed from a qualifier and a method that gets associated with that qualifier.

```
var avColorChangedVariant = ao.Advice.add({
//id       : "colorChangedVariant", // if omitted UUID will be generated
  type     : "around",
  handler  : function (proceed, handler, args, target/*, joinpoint*/) {
    var evt = args[0];
    evt.isColorChanged = true;
    return proceed.call(target, evt);
  }
});
```

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

Aspect

- An `Aspect` in JavaScript needs to feature just a sole function that enables folding of advices and pointcuts within it's function body.
- Thus an aspect has to be constructed from a callback function that's first argument is a method that links advices to pointcuts and that's second argument references the AO System itself.

```
var asColorChangedVariant = ao.Aspect.add({
//id      : "colorChangedVariant", // if omitted UUID will be generated
  handler : function (linkAdviceToPointcut, ao) {
    linkAdviceToPointcut(avColorChangedVariant, pcIsAnyToSubmit);
  }
});
```

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

Additional Notes

- In order to take advantage of JavaScripts dynamic nature it should be allowed to alter the whole system's control flow at any time from any point e.g.
  - advices do alter the system's control flow just by calling one of every advices two methods either `confirm` or `deny`.
  - add or remove joinpoints, pointcuts, advices regardless of how many aspects are currently *confirmed* or *denied*.
  - switching the whole AO System `off` and `on` again.

Thoughts about how to adopt the Principles of AOP to JavaScripts Dynamic and Functional Nature

»modification.ao«

(just in order to get a glimpse of its API)

```
var ao = require("modification.ao") // aspect oriented system

// static properties

ao.Joinpoint                    // [Object] // module
ao.Pointcut                     // [Object] // module
ao.Advice                       // [Object] // module
ao.Aspect                       // [Object] // module

// static methods

ao.isOff                        // [Function]:true|false
ao.isOn                         // [Function]:true|false
ao.off                          // [Function]:void
ao.on                           // [Function]:void
ao.reboot                       // [Function]:void
```

Questions?

Thank You

PDF Handout