Design Patterns for Scalable JavaScript Application by Legal-Box

Starting from the Scalable Javascript Application Architecture by Nicholas Zakas [1], this document gets one level deeper, describing the design patterns useful to implement it.

The document is intended for a technical audience.

Executive Summary

The design patterns proposed to implement the Scalable JavaScript Application Architecture fall into two different categories: general Design Patterns from the Gang of Four [2] and Design Patterns specific to JavaScript.

General Design Patterns

- **Proxy** [3] to impose strict rules on modules ; the sandbox objects act as proxies between user interface modules and the application core Facade
- **Facade** [4] to provide a single entry point to the methods part of the application core programming interface
- **Mediator** [5] to ensure loose coupling, all communication between user interface modules is channeled through the application core Facade. Through this link, modules send messages to the application core Facade, never to each other.
- **Observer** [6] mechanism provided by the application core Facade lets modules subscribe to events to get notified when these events occur. Each user interface module is an Observer of events broadcast by the application core Facade.
- **Adapter** [7] to translate messages from the objects used in the JavaScript application to the format used for communication with the server, and vice versa

Design Patterns Specific to JavaScript

- **JavaScript Module Pattern** [8] to create a distinct scope and namespace for each module, while keeping the global scope clean of all but one global object
- **Portlet** [9] user interface modules designed as independent applications keeping track of their independent state
- **Cross-Browser Component** [10] as a strong foundation for the architecture, to abstract differences between browsers

Document Metadata

Author: Eric Bréchemier

Version: 2011-04-11

Introduction

The initial version of this document, circa April 2010, documented the design study in preparation of the development of the Scalable Web Application framework at Legal-Box.

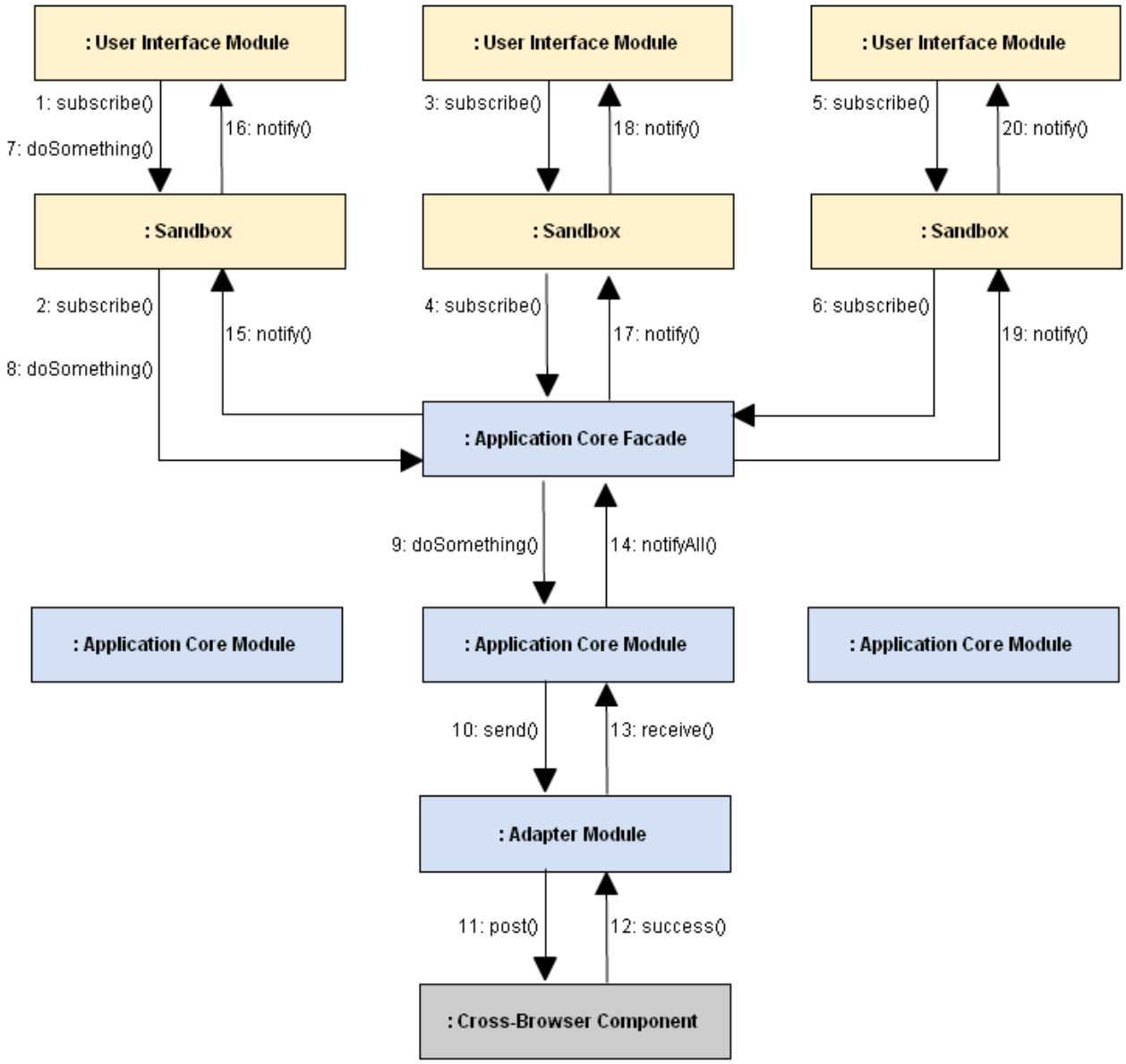
In the course of this development, the design shifted slowly apart from this design, while keeping true to the underlying principles.

In this new version (2010-05-07), I have chosen to keep the description of the initial design, while describing for each section how the actual implementation differs. You will find below one section describing the collaboration overview in the initial design, followed with a section explaining the differences in the actual implementation. In the second part of the document, “JavaScript Application Components”, I followed the same pattern at the level of subsections.

It is my hope that this presentation will convey a revealing perspective on the choices I made during the design and development of the Scalable JavaScript Application.

Collaboration Overview (Initial Design)

The UML collaboration diagram below exemplifies how the communication between user interface modules is channeled through the application core.

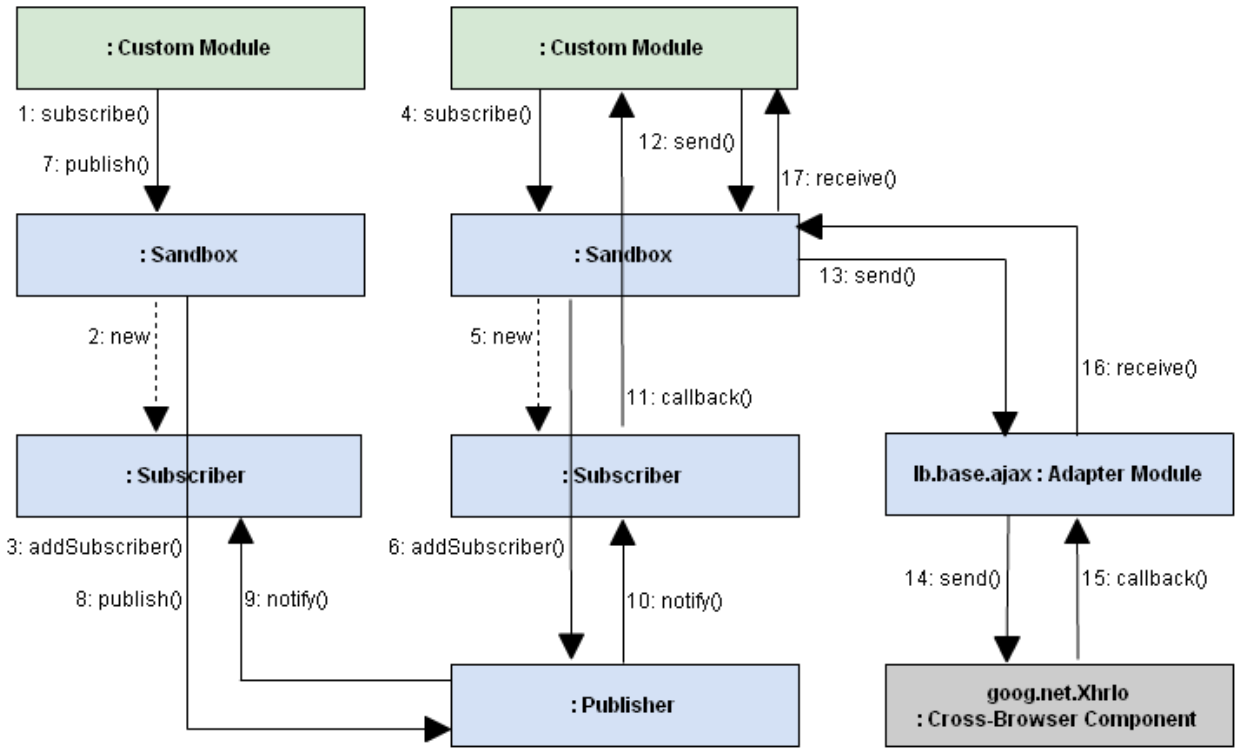


Sequence of Messages:

- 1-6: each user interface module subscribes through the Sandbox to events published by the Application Core Facade (*Proxy and Observer patterns*)
- 7-9: one user interface module calls through the Sandbox one of the methods of the Application Core API, which triggers a corresponding processing in one application core module (*Proxy, Mediator and Facade patterns*)

- 10-11: the Application Core Module sends a request to the server, which is translated by the Adapter into a suitable format, and processed by a Cross-Browser Component (*Adapter and Cross-Browser Component patterns*)
- 12-13: the asynchronous response from the server is translated by the Adapter into a JavaScript object and provided to the Application Core Module for further processing. (*Adapter and Cross-Browser Component patterns*)
- 14: the Application Core Module publishes a new event on the Application Core Facade (*Observer pattern*) as a result of its processing
- 15-20: each user interface module that subscribed to this event gets notified through its Sandbox, and updates its display in a completely independent way (*Proxy, Observer and Portlet patterns*)

Collaboration Overview (Actual Implementation)



In the actual implementation, there is no clear cut difference between User Interface Modules and Application Core Modules. All modules are just custom modules: specific code following severe guidelines by channeling all its interactions with the rest of the application through a dedicated Sandbox instance.

In this new collaboration schema, the Application Core Facade has been replaced with the Publisher object, which is responsible for the management of subscribers and the broadcasting of published events. There is a single Publisher for the whole application.

Instead of registering itself on the Facade, the Sandbox now creates a new Subscriber instance for each subscription. This Subscriber is initialized with a filter; only matching events will trigger the associated callback function.

Each Sandbox instance is now a Facade on its own; it handles interactions with adapter modules, e.g. lb.base.ajax for asynchronous communication with the host server.

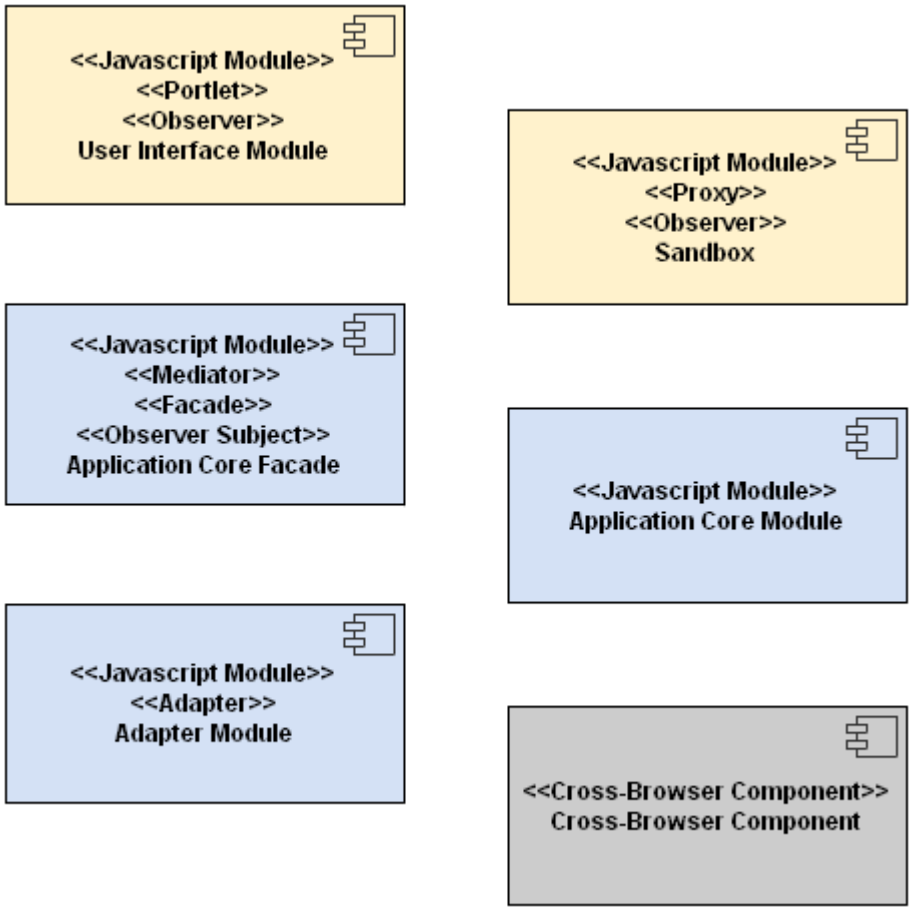
Sequence of Messages:

- 1-6: each custom module subscribes through its Sandbox to events published by the Core Events Publisher (*Proxy and Publisher/Subscriber patterns*)
- 7-8: the first module publishes a new event through its Sandbox (*Proxy and Publisher/Subscriber patterns*)
- 9-11: the new event is broadcast by the Publisher to all Subscribers. The first module, which did not include this type of event in its subscription filter, is not notified. The second module was interested in this kind of event, and it gets notified through configured callback. (*Proxy and Publisher/Subscriber patterns*)
- 12-14: the second module is responsible for posting update messages to the host server. It sends a new message through its Sandbox, which gets serialized and transferred to the server. (*Proxy, Adapter and Cross-Browser Component patterns*)
- 15-17: when the asynchronous answer from the server is received, the callback provided in the request, receive(), is triggered, and the module gets the answer as a JavaScript object (*Proxy, Adapter and Cross-Browser Component patterns*)

JavaScript Architecture Components

The UML diagram below summarizes the different types of components present in the JavaScript Application Architecture. I annotated each type of component with the corresponding Design Patterns, using <<stereotypes>>.

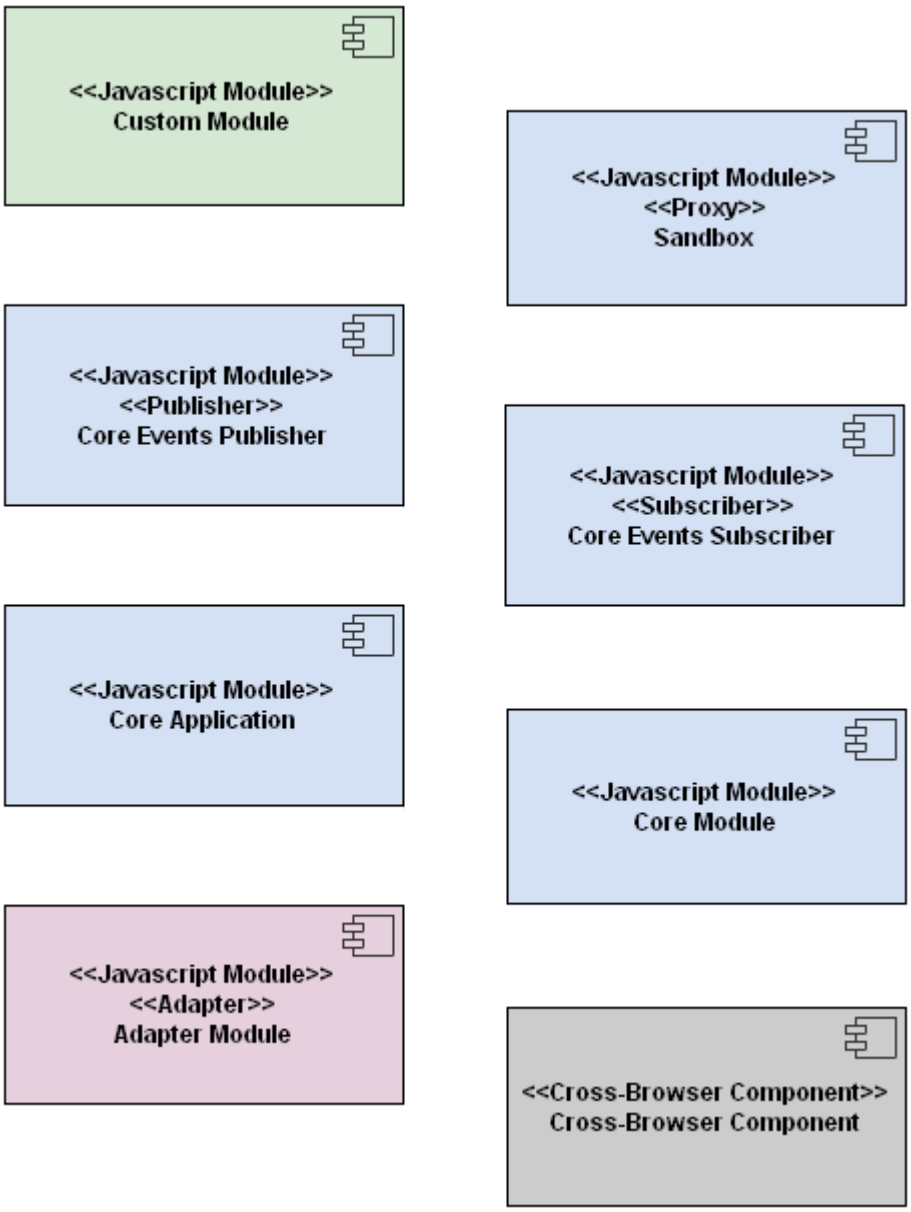
Initial Design



In the initial design,

- User Interface Modules are expected to be different and distinct from Application Core Modules
- the Sandbox is part of the User Interface layer
- the Application Core Facade holds many responsibilities

Actual Implementation



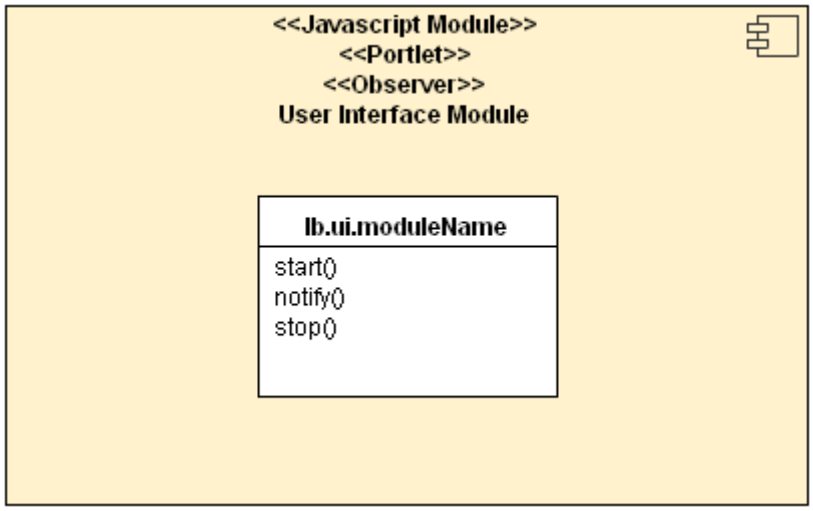
In the actual implementation,

- all modules are custom modules, whether they are intended to handler interactions in User Interface or in Data Model.
- the Sandbox is now part of the core layer and interacts directly with adapter modules and other parts of the core.
- instead of a single Facade with many responsibilities, a Core Application in charge of managing modules life cycle and a Publisher responsible of events broadcasting
- Adapter Modules are considered a separate layer, intended for portability over different base libraries of cross-browser components.

(Initial Design)

Intended as a guideline for the design of these components, each type of component is described in its own section below.

Design of User Interface Modules (Initial Design)



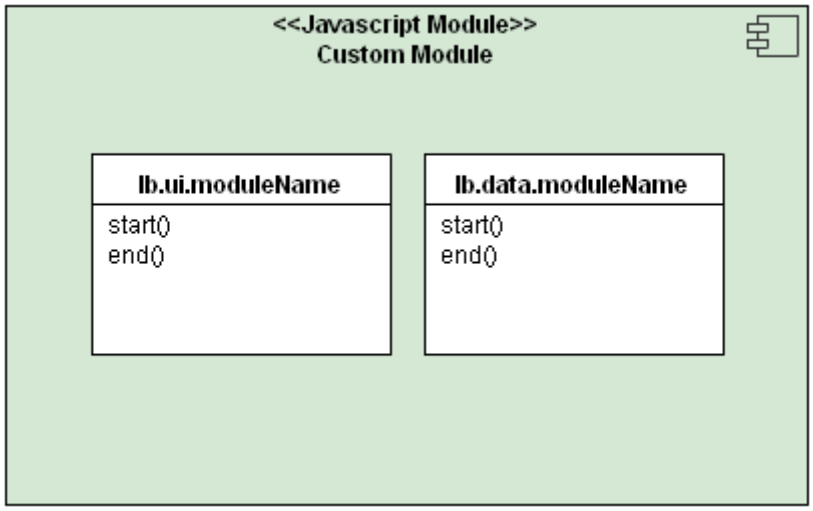
Each module shall live in its own namespace, by using the JavaScript Module Pattern. Let's call the top-level namespace lb (for Legal-Box). Within lb, two different namespaces will differentiate User Interface modules in lb.ui from Application Core modules in lb.core.

Following JavaScript convention, User Interface module names start with a lower-case letter because they are closure creator functions, not constructor functions to be called with new.

To clarify the intent in current design, I chose to rename the init() and destroy() functions in Nicholas Zakas' examples to start() and stop(). Similarly, I renamed the handleNotification() function to notify() for consistency with the Observer Pattern.

Each module may define any specific method needed for its own use. These methods shall remain in its private scope.

Design of Custom Modules (Actual Implementation)

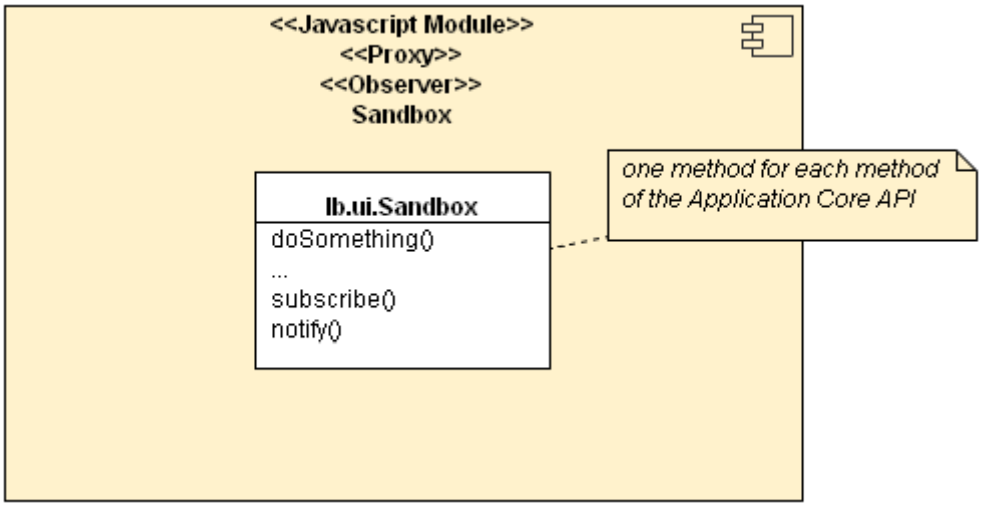


In the actual implementation, two kinds of modules are expected to behave in the same way: User Interface Modules, living in the namespace lb.ui, and Data Model modules, in the namespace lb.data.

The difference between User Interface modules and Data Model modules may get fuzzier: it is possible that some module end up having both the responsibility of handling user interactions and keeping a record of underlying data objects specific to the domain.

The notify() method has been removed from the expected interface, – it is now a callback provided as parameter in subscribe() –, and the stop() method has been renamed to end() to clarify the intent: it is expected to terminate the life cycle of the module, this is not a mere pause of module interactions. Both start() and end() are optional.

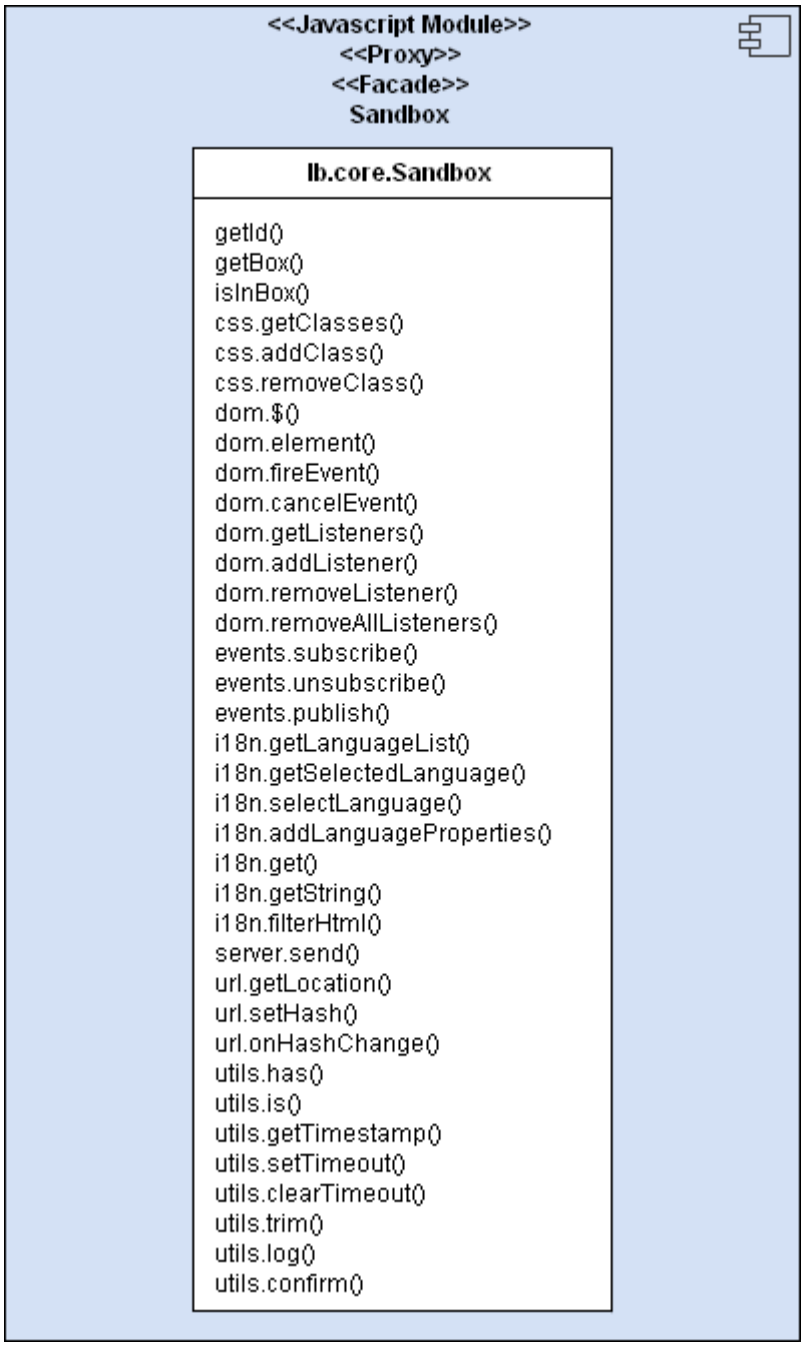
Design of the Sandbox (Initial Design)



For the purpose of enforcing security, the Sandbox as defined by Nicholas Zakas acts as a Proxy between User Interface modules and the Application Core Facade. As such, the Sandbox must expose each public method of the Application Core API. The Sandbox shall be updated to reflect any change in this API.

Besides, the Sandbox acts as a filter in the communication of each User Interface module with the rest of the application. Using the Observer pattern implemented with the subscribe() and notify() methods allows to have loose coupling between each module and the application.

Design of the Sandbox (Actual Implementation)



In the actual implementation, the Sandbox is the sole provider of the core API, and acts both as a Proxy and a Facade for the Core Application. Providing a separate instance to each module avoid naughty bugs by pollution of the API [16].

There are 36 methods available on the Sandbox, falling into 8 categories; each category is defined as a property which provides an intermediate step to group methods of similar purpose:

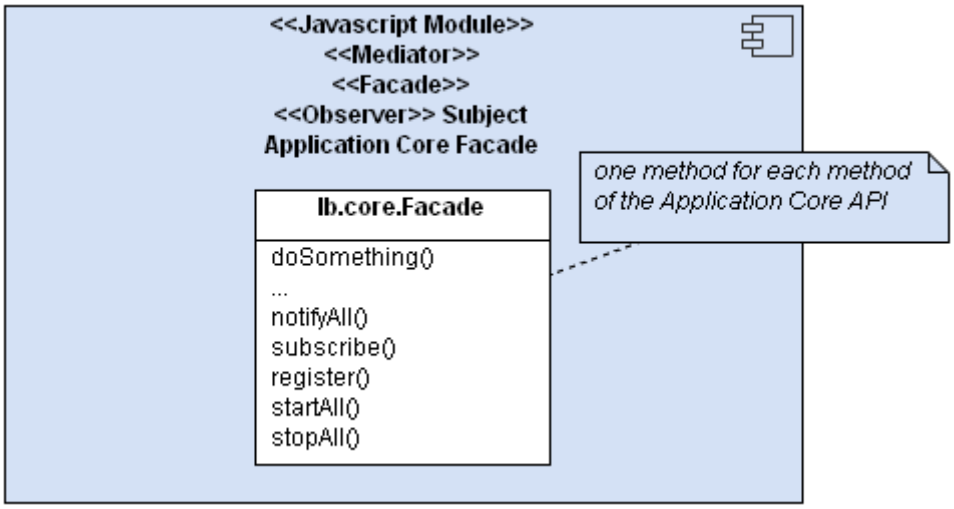
1. (no prefix) – properties of this module, the identifier and box
    - sandbox.getId()
    - sandbox.getBox()

    - sandbox.isInBox()
2. css – Cascading Style Sheets utilities
    - sandbox.css.getClasses()
    - sandbox.css.addClass()
    - sandbox.css.removeClass()
3. dom – Document Object Model utilities
    - sandbox.dom.$()
    - sandbox.dom.element()
    - sandbox.dom.fireEvent()
    - sandbox.dom.cancelEvent()
    - sandbox.dom.getListeners()
    - sandbox.dom.addListener()
    - sandbox.dom.removeListener()
    - sandbox.dom.removeAllListeners()
4. events – for loose coupling with other modules
    - sandbox.events.subscribe()
    - sandbox.events.unsubscribe()
    - sandbox.events.publish()
5. i18n – for internationalization
    - sandbox.i18n.getLanguageList()
    - sandbox.i18n.getSelectedLanguage()
    - sandbox.i18n.selectLanguage()
    - sandbox.i18n.addLanguageProperties()
    - sandbox.i18n.get()
    - sandbox.i18n.getString()
    - sandbox.i18n.filterHtml()
6. server – for asynchronous communication with a remote server
    - sandbox.server.send()
7. url – URL and local navigation
    - sandbox.url.getLocation()
    - sandbox.url.setHash()
    - sandbox.url.onHashChange()
8. utils – general utilities
    - sandbox.utils.has()

- sandbox.utils.is()
- sandbox.utils.getTimestamp()
- sandbox.utils.setTimeout()
- sandbox.utils.clearTimeout()
- sandbox.utils.trim()
- sandbox.utils.log()
- sandbox.utils.confirm()

As a Proxy, the Sandbox hides details of the implementation: for example, the DOM methods check that target elements are part of the box assigned to the module, and methods to create elements, listeners and events on DOM elements work in collaboration with a customizable factory, which may extend the DOM elements with enhanced behaviors to provide the experience of Rich Internet Application Widgets.

Design of the Application Core Facade (Initial Design)



The design of the Application Core Facade crystallizes many different facets into a single module.

1. It provides methods to manage the life cycle of modules:

- register() to register (declare) a new module
- startAll() to start all registered modules at the start of the web application
- stopAll() to stop all started modules at the end of the web application

To simplify, I chose to remove the start() and stop() functions from these public methods. They are now functions of each module.

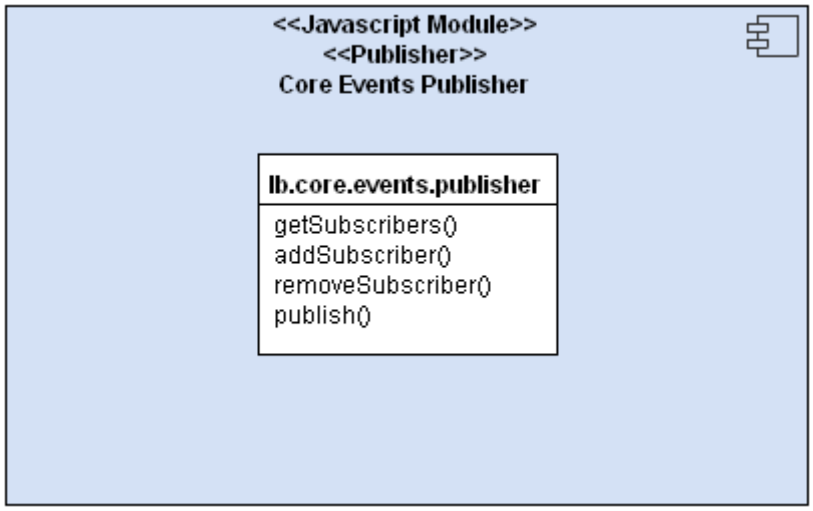
2. For the purpose of loose coupling, it manages an event broadcasting system:

- subscribe() to start a subscription to some kind of events
- notifyAll() to broadcast an event to all subscribed Observers

Note: there is currently no method to stop a subscription. It may be added if needed.

3. It provides a single entry point for the Application Core Programming Interface. The details of this API are outside of the scope of the current document.

Design of the Events Publisher and Subscribers (Actual Implementation)



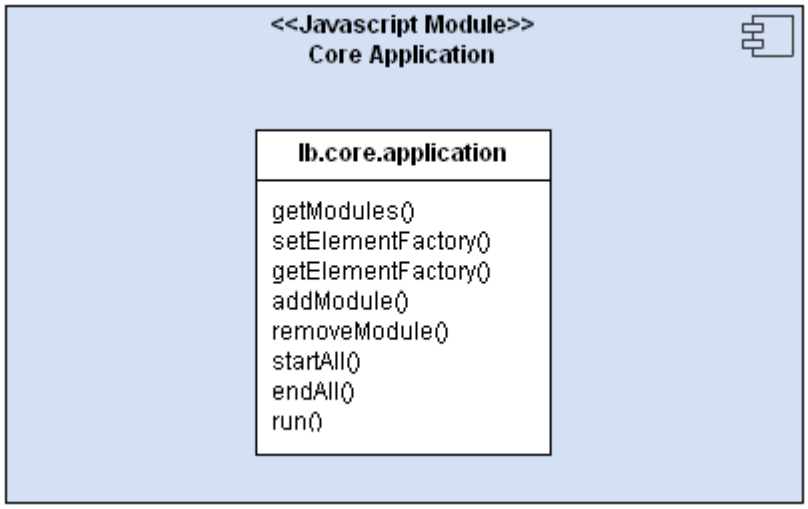
In the actual implementation, the Observer pattern was replaced by a similar Publisher/Subscriber Design Pattern.

The publisher (a single instance for the application) manages subscribers and broadcasts published events to all subscribers.

The Subscriber (one instance for each filter/callback group, there might be several for a single module) filters incoming events and triggers a callback if needed.

An important feature of the Subscriber in the actual implementation is to clone incoming events before propagation to custom modules. It avoids undesired interactions between modules through shared objects [16] and allows each module to keep and modify the event data freely, event after the broadcasting of the event completes.

Design of the Core Application (Actual Implementation)



The Core Application was not part of the initial design. It is a fork from parts of the Application Core Facade.

It is in charge of managing the life cycle of modules. The Core Application is intended to be used by custom applications to manage a set of custom modules:

- the custom application creates and adds modules of its choice to the Core Application
- a call to run() registers startAll() on the page load to start all modules, and endAll() on the page unload event to terminate all modules.

Two methods appear in the Core Application in this version (2010-05-18), designed as an extension point for Rich Internet Application Widgets:

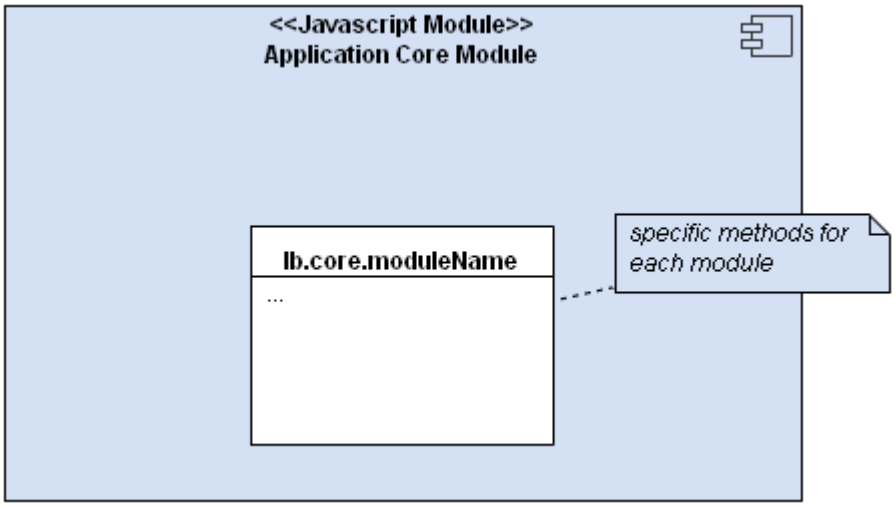
- setFactory() to configure a new factory for DOM elements, listeners and events
- getFactory() used by Sandboxes to retrieve the configured factory

While the default factory simply creates regular DOM elements, a custom factory would typically initialize widgets on top of these elements. For example, in a custom factory, all ul/ol elements with the class “menu” can be enhanced with a drop-down menu behavior.

A custom factory must implement methods to create elements, listeners and events, as well as corresponding methods to destroy these objects at the end of life of the module.

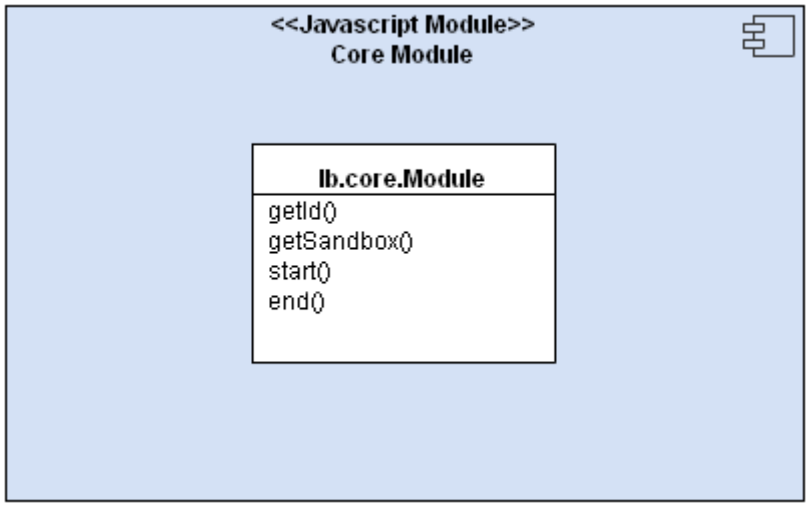
The communication with Rich Internet Application Widgets is based on custom events using the same Sandbox methods used for regular DOM events: addListener() to create and register a listener and fireEvent() to create and trigger an event, or cancelEvent() to prevent a default action associated with the event, e.g. the submission of a form.

Design of Application Core Modules (Initial Design)



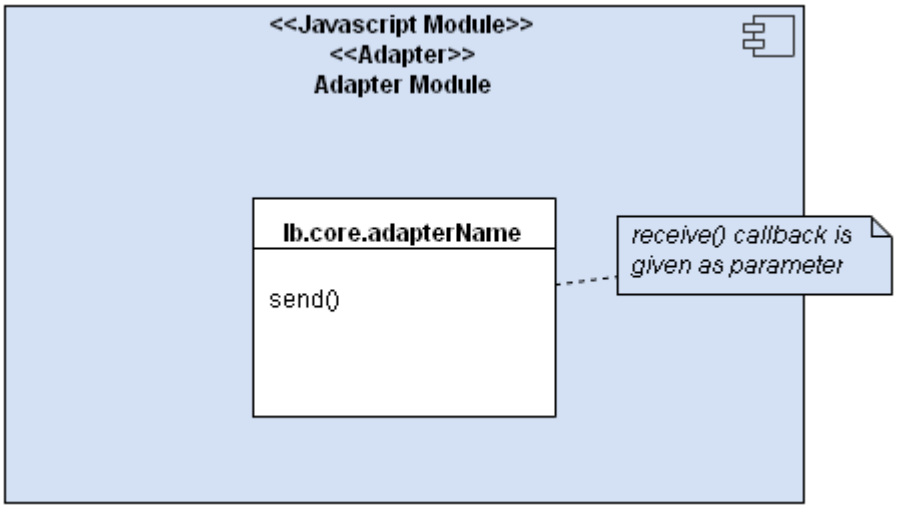
Application Core Modules shall define functions for all the functionalities of the Legal-Box system accessible from a browser. These functions are specific to Legal-Box system and will be reflected, in a somewhat summarized form, in the Legal-Box API exposed in the Application Core Facade.

Design of Application Core Modules (Actual Implementation)



The Core Application does not work directly with instances of custom modules, but through generic wrappers, the Core Modules, which handle errors and the absence of optional methods. On the contrary of the initial design, there are no methods specific to each module.

Design of Adapter Modules (Initial Design)



The responsibilities of the Adapter Modules are to abstract the details of the communication with the server, in terms of format and protocol used.

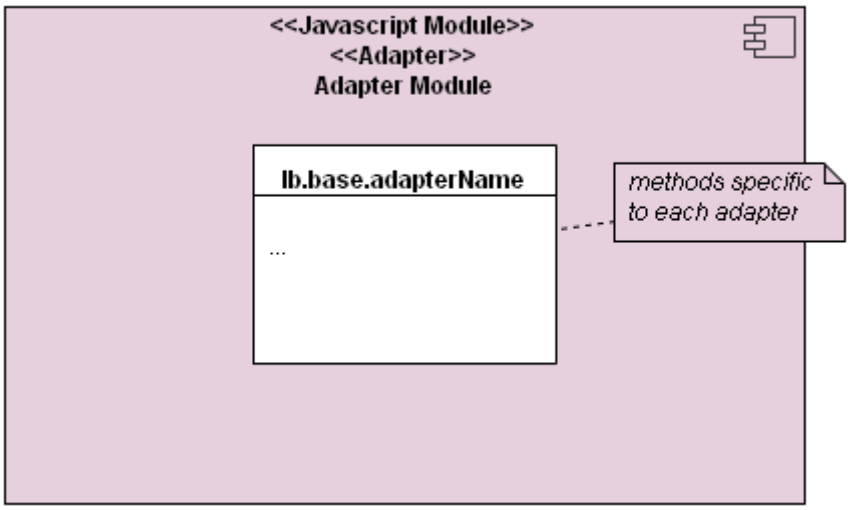
Using the Adapter pattern has the following advantages:

- defining our own methods lowers the coupling with the underlying Cross-Browser components used
- the data conversion step ensures that no module relies on the specifics of the format used for the communication with the server (e.g. XML vs JSON vs key/values in application/x-www-form-urlencoded or in multipart/form-data)

The send() method is intended for the communication of a single asynchronous message from the browser to the server, with a callback provided as parameter to provide the converted response.

Additional methods may be needed for different types of messages, e.g. to send a file as opposed to sending data.

Design of Adapter Modules (Actual Implementation)

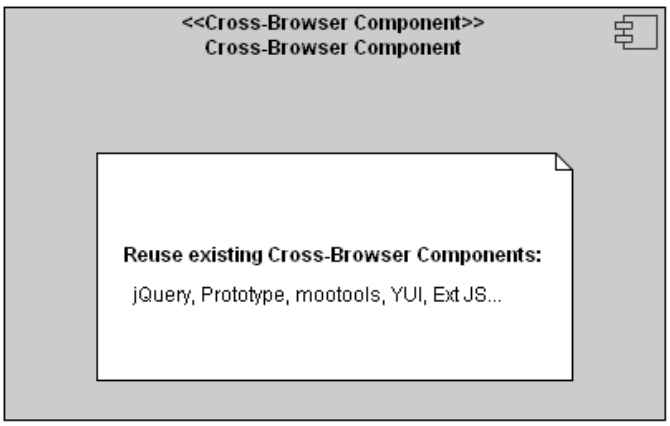


There are different kinds of adapter modules in actual implementation, related to different domains:

- AJAX communication with the host server
- Internationalization (i18n)
- JSON parsing and serialization
- DOM manipulations
- Logging
- HTML and Text Templates
- Utilities for native JavaScript: string, array, object

These adapters provide a portability layer over the chosen base library.

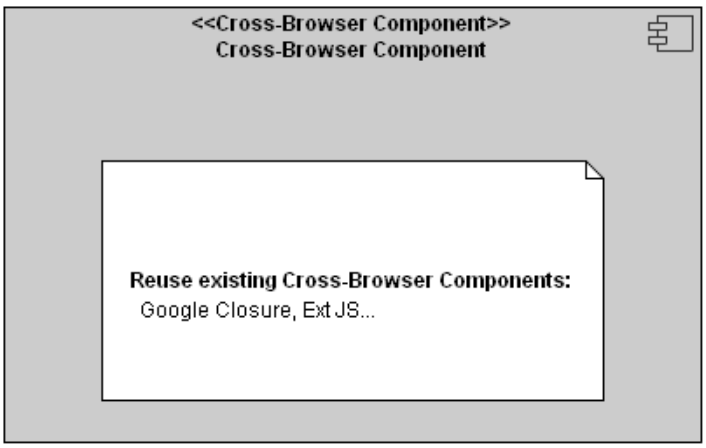
Do Not Design Cross-Browser Components (Initial Design)



We shall not design new Cross-Browser components because:

- reliable Cross-Browser components are widely available [11] [12] [13] [14] [15]
- the cost of development and maintenance for such components is very high

Do Not Design Cross-Browser Components (Actual Implementation)



I actually did not design any cross-browser component, but the selection process for a suitable base library was harder than expected. Although many comprehensive libraries of cross-browser components available indeed, most (if not all) of them pollute the global namespace, the prototype of native JavaScript objects or the DOM nodes they manipulate in some way or another.

I selected Google Closure [17] for its modular nature, its completeness, and its appearance of high quality. I extracted only the files of interest with the help of a dependency calculation tool distributed with the library.

Still, I had to edit the base file to move some global configuration parameters to the goog namespace.

Conclusion and Directions for Further Work (Initial Design)

This document lays the ground foundation for the implementation of the Scalable JavaScript Application Architecture by Nicholas Zakas [1].

From this point,

- the Application Core API specific to Legal-Box system must be specified
- Cross-Browser components must be selected based on requirements

to fill the gaps in current design.

Conclusion and Directions for Further Work (Actual Implementation)

With the current implementation of the framework, there is no need for the design of a complete Application Core API specific to Legal-Box. Custom modules can be designed iteratively, and define events of interest.

From this point,

- a set of guidelines should be defined for the format of event objects: naming conventions and semantic for a lingua franca
- cross-browser Rich Internet Application components should be integrated in the framework, by creating useful abstractions in adapters over a base library such as Ext JS [18]

References

[1] *Scalable JavaScript Application Architecture* by Nicholas Zakas
http://www.slideshare.net/nzakas/scalable-javascript-application-architecture

[2] *Design Patterns: Elements of Reusable Object-Oriented Software* by Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
http://en.wikipedia.org/wiki/Design_Patterns

[3] *Proxy Pattern* – from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Proxy_pattern

[4] *Facade Pattern* – from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Facade_Pattern

[5] *Mediator Pattern* – from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Mediator_Pattern

[6] *Observer Pattern* – from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Observer_Pattern

[7] *Adapter Pattern* – from Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Adapter_Pattern

[8] *JavaScript Module Pattern: In-Depth* by Ben Cherry
http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth

[9] *Portlet* by Michael Mahemoff
http://ajaxpatterns.org/Portlet

[10] *Cross-Browser Component* by Michael Mahemoff
http://ajaxpatterns.org/Cross-Browser_Component

[11] *jQuery: The Write Less, Do More, JavaScript Library*
http://jquery.com/

[12] *MooTools - a compact javascript framework*
http://mootools.net/

[13] *The Dojo Toolkit - Unbeatable JavaScript Tools*
http://www.dojotoolkit.org/

[14] *Prototype JavaScript framework: Easy Ajax and DOM manipulation for dynamic web applications*
http://www.prototypejs.org/

[15] *YUI Library*
http://developer.yahoo.com/yui/

[16] *Maintainable JavaScript: Don't modify objects you don't own* by Nicholas Zakas
http://www.nczonline.net/blog/2010/03/02/maintainable-javascript-dont-modify-objects-you-down-own/

[17] *Closure Library – Google code*
http://code.google.com/closure/library/

[17] *Ext JS — JavaScript Framework and RIA Platform*
http://extjs.com/