Opening Personalization to Partners: An Architecture of Participation for Websites

Cristóbal Arellano, Oscar Díaz, and Jon Iturrioz

ONEKIN Research Group, University of the Basque Country (UPV/EHU),
San Sebastián, Spain
{cristobal.arellano,oscar.diaz,jon.iturrioz}@ehu.es
http://www.onekin.org/

Abstract. Open innovation and collaborative development are attracting considerable attention as new software construction models. Traditionally, website code is a "wall garden" hidden from partners. In the other extreme, you can move to open source where the entirety of the code is disclosed. A middle way is to expose just those parts where collaboration might report the highest benefits. Personalization can be one of those parts. Partners might be better positioned to foresee new ways to adapt/extend your website based on their own resources and knowledge of their customer base. We coin the term *"Open Personalization"* to refer to those practises and architectures that permit partners to inject their own personalization rules. We identify four main requirements for OP architectures, namely, resilience (i.e. partner rules should be sheltered from website upgrades, and vice versa), affordability (easy contribution), hot deployment (anytime rule addition), and scalability. The paper shows the approach's feasibility using *.NET*.

Keywords: Personalization, Open Development, .NET, MEF.

1 Introduction

Web personalization refers to making a Web site more responsive to the unique and individual needs of each user [4]. It accounts for important usability and productivity gains, specifically for organizational websites. Here, it is important to notice that organizations seldom work in isolation. Organizations establish (contractual) relationships with their partners to achieve their goals. Suppliers, collaborators, associates and the like are common terms to reflect these ecosystems. Hence, it is just natural that these relationships commonly surface the website of these organizations. Corporate websites tend to include data about the logistics, payment or providers, which do not represent the kernel of the corporate activity but collaborate to fulfil the corporate's objectives. Even an ephemeral activity such as a conference organization includes in its website, data about hotel partners, proceeding publishers or sponsors which might all be subject to contractual agreements. In this setting, this work

M. Brambilla, T. Tokuda, and R. Tolksdorf (Eds.): ICWE 2012, LNCS 7387, pp. 91–105, 2012.

addresses the following research question: *How is Web personalization affected by the collaborative nature of the organization activities to which the website gives support to?*

Traditional personalization assumes a centralized approach. The website master (the "who") decides the personalization rules (the "what"), normally at the inception of the website (the "when"). In this context, partners tend to be mere stakeholders who do not actively participate in the *development* of the website. However, personalization opportunities might be difficult to foresee by the website master. Indeed, as documented in [2], a large rate of interesting innovations comes from the users/partners once the system is in operation. This scenario is also akin to *open innovation* [8], and the *client-shared-source software model* where vendors let partners access the source code through a common platform [14]. By its very nature, personalization is a perfect candidate for being subject to "open innovation". In addition, resource scarcity makes the website master only incorporate major enhancements while a more active participation of the partners could also serve the long tail.

Therefore, we want custom extensions to be built by any partner instead of being left only to the web master. We introduce the notion of ***Open Personalization (OP)*** as a means for partners to collaborate in the personalization of the website. The premise is that owners might be willing to open their websites provided (1) minimal additional burden is required and (2), ability of partners to contribute with valuable and up-to-date content for the website users (even if outside the website business model). OP might lead to new business models where openness might be subject to agreements on how to split potential revenues similar to the way *Google AdWords* works. This model can be of interest when partner relationships surface the website of the host. This includes online portals that offer third-party products such as travel agencies (with partnership with resorts and air carriers) or department stores (with partnership with logistic companies).

This paper's main contribution rests on proving the technical feasibility of such approach by introducing an OP architecture for *.NET*. First, we identify a set of quality criteria for OP architectures (Section 3). Next, we address the realization of OP from the partners' perspective (i.e. definition of their own personalization strategies) and the host viewpoint (i.e. how to safely disclose code) in Sections 4 and 5, respectively. Section 6 revises the OP architecture along the requirements previously set. We start by confronting "closed personalization" versus "open personalization".

2 "Closed Personalization" *versus* "Open Personalization"

Typically, Web design methods define three main models: the *Domain Model*, in which the structure of the domain data is defined; the *Navigation Model*, in which the structure and behaviour of the navigation view over the domain data is defined, and finally, the *Presentation Model*, in which the layout of the generated hypermedia presentation is defined. On these grounds, *personalization rules* are

defined that adapt any of the three models based on the characteristics of the current user. This implies the introduction of two additional models: the *User Model* (e.g. demographic data, relevant behaviour while interacting with the site, etc.) and the *Personalization Model*. Broadly, the *Personalization Model* commonly resembles that of condition-action rules. The condition basically checks the state of the *Domain Model* and the current *User Model*. The action impacts the navigation structure and presentation, and might also update the user information specified in the *User Model*.

Distinct commercial tools (e.g. ILog JRules, LikeMinds, WebSphere, Rainbow, Infusionsoft) help to define and manage the personalization strategy. These tools might play the role of frameworks (providing an enhanced container where to run your code) or IDEs (helping in generating the code). No matter the approach, the generated code commonly follows the *Model-View-Controller* pattern. For the case of .NET, the mapping goes as follows: (1) the Domain Model is realized as a set of C# classes, (2) the User Model is kept in a special class known as the *ProfileBase*; (3) the Navigation Model is supported through *Controller* classes which can check the Model classes (including *ProfileBase*) and decide which content to pass to the View through *ViewData*, a system-defined variable for Controller-View data passing; (4) the Presentation Model is realized as a set of Web Forms which provide the appropriate renderings for the data kept in *ViewData*. In this setting, a personalization rule commonly ends up being realized as part of the Controller, and impacting the View.

As an example, consider the ICWE'09 conference website. The website basically contains standard information for conferences, i.e. papers, keynotes, accommodations, etc. It is a one-size-fits-all solution where all attendees get the very same content. We have extended the original site with login so that role-based personalization is now possible based on whether the current user is a PC member, a session chair or an author. For instance, additional banquet information can be displayed when login as an attendee with a full passport. This example illustrates "closed personalization": the Web administrator (the "who") decides the personalization rules (the "what"), normally at the inception of the website (the "when"). More sophisticated approaches such as those based on configurations or detection of access patterns (i.e. adaptive and adaptable techniques [3]) are a step ahead but they are still centrally foreseen and developed by the host designer. Of course, partners can participate as stakeholders, and contribute with some personalization scenarios. Some examples follow for the ICWE website:

- Barceló Resorts FACILITATES a 50% discount on room booking over the weekend, PROVIDED the attendee holds a full passport,
- Springer-Verlag FACILITATES a 10% discount on books authored by the seminars' speakers, PROVIDED the attendee is registered for this seminar,
- The Tourism Information Office FACILITATES information about cultural activities on the city during the free slots left by the conference program.

Supporting (and maintaining) these scenarios still rests on the host's shoulders. This setting is not without bumps. First, owner's lack of motivation. The website

owner might regard previous scenarios as not aligned with its business model (e.g. room offers might not attract more conference attendees) and hence, not paying-off the effort. Second, partnership might be dynamic, being set once the website is in operation (e.g. pending agreements with the publisher). For instance, the aforementioned rule by Springer-Verlag might require updating not just the View but also the Controller, and even the User Model if seminar attendance is not recorded. As a result, partner rules might end up not being supported by the website. This is not good for any of the actors. End users lose: they will not get the discounts or overlook interesting data. Partners lose: they miss an opportunity to drive more customers to their services. Website owners lose: the website reduces its "stickiness", missing the chance to become a true data hub for the subject at hand (e.g. the ICWE conference).

Open Personalization (OP) pursuits to engage external partners in the personalization endeavour: partners introduce their rules on their own with minimal impact on the owner side. This arrangement makes more economical sense. Partners might regard OP as a chance to increase their own revenues by personalizing their offerings in those websites that serve as a conduit for their products/services (e.g. room offers when booked through the conference website). On the other side, the owner can be willing to facilitate (rather than develop) such initiatives for the good of its customers as long as its involvement is limited. However, OP should not be viewed only as a way to share the maintenance cost but as an enabler of and means for truly collaborative solutions and lasting partner relationships. In this paper however, we focus on the technical feasibility of OP.

3 Open Personalization: Requirements

Open APIs are one of the hallmarks of the Web2.0 whereby Web applications disclosure their data silos. However, "opening data" is not the same that "opening personalization". Personalization requires not only access to the data but also adaptation in the content/navigation/layout of the website. OP would then mean to offer (controlled) access to the *User/Domain Model* (better said, their implementation counterparts) and the (regulated) introduction of the partners' personalization rules (hereafter referred to as ***"mods"***). This basically calls for "an architecture of participation". This term was coined by Tim O'Reilly "to describe the nature of systems that are designed for user contribution" [12]. O'Reilly writes that "those that have built large development communities have done so because they have a modular architecture that allows easy participation by independent or loosely coordinated developers". OP is then about creating a community with your partners.

Based on these observations, we introduce the following quality criteria (and driven requirements) for "an architecture of participation" for OP:

- **Resilience**. *Mods* should be shelter from changes in the underlying website, and vice versa, partners' code should not make the website break apart.

- **Extensibility**. OP departs from some model-driven approaches where personalization is decided at design time and captured through models. Mods can be added/deleted as partnership agreements change throughout the lifetime of the website.
- **Scalability**. Growing amount of *mods* should be handled in a capable manner.
- **Affordability**. Partner effort should be minimized. Designs based on widely adopted programming paradigms stand the best chance of success. Intricate and elaborated programming practices might payoff when used internally, but the advantage can be diluted when partners face a steep learning curve. The more partners you expect to attract, the simpler it must be and the more universal the required tools should be.

As a proof of concept, next section introduces "an architecture of participation" for .NET driven by the aforementioned requirements.

4 Open Personalization: Specification

OP is about disclosing code for partners to inlay their *mods*. Therefore, we risk existing *mods* to fall apart when the underlying website is upgraded (i.e. the code changes), hence putting an additional maintenance cost on partners. Isolation solutions should be sought to ensure that the evolution of the website has minimal impact on the existing *mods*. Among *.NET* artefacts (i.e. the Model classes, the Web Forms and the Controller classes), Model classes are certainly the most stable part of a Web application. Therefore, *mods* pivot around Model classes. Those classes that are amenable to participate in a *mod* are said to support a **Modding Concept**.

> *A Modding Concept is a Model Class whose rendering realization (i.e. Web Forms) is amenable to be leveraged by a partner through a* ***mod****,* i.e. *an HTML fragment to be injected into the appropriate Web Forms.*

The latter still suggests that *mods* might be affected by changes in Web Forms. To ensure decoupling, all interactions between Web Forms and *mods* are conducted through events. Model classes are manipulated through traditional set/get methods. In addition, those classes playing the role of Modding Concepts have an additional interface, the **Modding Interface**, which holds[1]:

- **Publishing Events**, which notify about instances of Modding Concepts (e.g. *Accommodation*) being rendered by the website. For instance, the event *LoadAccommodation* is produced by the host everytime an accommodation is rendered. This event can be consumed by a *mod* through a handler (a.k.a. listener).

[1] The terminology of "processing events" and "publishing events" is widely used for event-based components such as portlets [10].

```
1  using System.Collections.Generic;
2  [ModdingConcept(PublishingEventType.Load)]
3  [ModdingConcept(ProcessingEventType.AddViewMod,"HTMLTableCellElement")]
4  public class Accommodation : IAccommodation {
5    [ModdingProperty]
6    public string Name { get; set; }
7    public string Url { get; set; }
8    [ModdingProperty]
9    public int Stars { get; set; }
10   [ModdingProperty]
11   public double SinglePrice { get; set; }
12   [ModdingProperty]
13   public double DoublePrice { get; set; }
14   public double Distance { get; set; }
15   public bool Breakfast { get; set; }
16 }
17 [ModdingConcept(PublishingEventType.Load)]
18 public class Profile : IProfile {
19   [ModdingProperty]
20   public string UserName { get { /*...*/ } }
21   public string FirstName { get { /*...*/ } }
22   public string FamilyName { get { /*...*/} }
23   public string Email { get { /*...*/ } }
24   [ModdingProperty]
25   public string RegistrationType { get { /*...*/ } }
26   [ModdingProperty]
27   public IEnumerable<ITutorial> PlansToAttendTutorial { get { /*...*/ } }
28   [ModdingProperty]
29   public IEnumerable<IRole> HasRoles { get { /*...*/ } }
30 }
```

Fig. 1. Domain classes annotated to become *Modding Concepts*

– **Processing Events** (a.k.a. actions), which are those that output an HTML fragment. For instance, the event *AddViewModAccommodation* provides a HTML fragment to be injected in those places where *Accommodation* instances are rendered. Therefore, *mods* can decide *what* to add but not *where* to add it. The latter is up to the host. For instance, the *AddViewModAccommodation* event is produced by a *mod* but it is let to the host decide where to handle it.

This notion of Modding Concept aims at minimizing the impact of OP for owners and partners alike. This is the topic of the next subsections.

4.1 Impact on the Host: Making a Website Mod-Aware

The additional effort required for a traditional website to become mod-aware is: (1) annotating the Model classes and (2), introducing place holders to locate *mod* output in Views (i.e. Web Forms).

Annotating Model Classes. Model classes can be decorated with the annotation *[ModdingConcept]*. Figure 1 shows the case for the ICWE website: the class *Accommodation* becomes a Modding Concept. *[ModdingConcept]*

```
1  <%@ Page Language="C#" MasterPageFile="~/Views/Shared/ICWE.master" %>
2  <asp:Content ContentPlaceHolderID="contentPlaceholder" Runat="Server">
3    ...
4    <% foreach (Accommodation acc in (IList<Accommodation>)ViewData["Accommodations"]) { %>
5      <tr>
6        <td class="text"><%:acc.Name%> <%:acc.Stars%>*<br/>...</td>
7        ...
8        <%=(((Dictionary<Accommodation, String>)ViewData["AddViewModAccommodation"])[acc])%>
9      </tr>
10   <%} %>
11   ...
12 </asp:Content>
```

Fig. 2. Mod-aware Views: the ASPX includes a place holder that accesses the *AccommodationMod* (line 8)

annotations produce Modding Interfaces. These interfaces are termed after the annotated class (e.g. the *Accommodation* class will generate the *IModdingConceptAccommodation* interface). This interface collects all the events to mod *Accommodation*. Event names are obtained from the event type (*Load*) plus the class name as a suffix (e.g. *LoadAccommodation*, *AddViewModAccommodation)*. Each annotation introduces an event type. So far, publishing events are limited to *"Load"* whereas processing events include *"AddViewMod"*. The latter outputs an HTML fragment hence, its payload is HTML-typed [15]. For instance, modding an *"Accommodation"* is set to be of type *HTMLTableCellElement,* meaning that mods to *Accommodation* need to be compliant with this type. This introduces a type-like mechanism for modding regulation. It can then be checked whether this *payloadType* is fulfilled, and if not so, ignores the mod but still renders the rest of the page. If *Accommodation* is rendered in different Views with different HTML requirements then, different *AddViewModAccommodation* events can be defined associated with distinct HTML types. It is also worth noticing that *not* all properties of a modding class might be visible. Properties available for *mods* are annotated as *[ModdingProperty]*.

Introducing Place Holders in Views. A View is mod-aware if it foresees the existence of mods that can produce additional HTML fragments to be inlayed in the View. This is so achieved using place holders. Commonly, Views that render Modding Concepts should cater for this situation, though this is up to the host. Figure 2 provides a View that renders *Accommodation* data. Since *Accommodation* is a Modding Concept, this View introduces a place holder (line 8). In *.NET*, data passing between the Controller and the View is achieved through the system variable *ViewData*. This variable holds an array for each possible type of data that can be passed. By convention, this array is indexed based on the type of the variable (e.g. *ViewData["Accommodations"]* conveys accommodations). Likewise, we use the convention of adding the prefix *"AddViewMod"* to the concept (e.g. *AddViewModAccommodation*) to refer to the information passed from the mod to the View (through the Controller). In this case, the content is an HTML fragment. The View retrieves this fragment, and

```
1  using System; using System.Collections.Generic; using System.Linq; using System.Text; using S
2  [InheritedExport]
3  public interface IPlugin {}
4  public class HotelPlugin : IPlugin {
5    IProfile profile; IList<IAccommodation> accommodations; bool done;
6    IModdingConceptProfile Profile; IModdingConceptAccommodation Accommodation;
7    [ImportingConstructor]
8    public HotelPlugin(IModdingConceptProfile i1, IModdingConceptAccommodation i2) {
9      Profile = i1; Accommodation = i2;
10     accommodations = new List<IAccommodation>(); done = false;
11     Profile.load += new EventHandler<LoadProfileEvent>(loadProfileHandler);
12     Accommodation.load += new EventHandler<LoadAccommodationEvent>(loadAccomodationHandler);
13   }
14   void loadProfileHandler(object sender, LoadProfileEvent loadProfileEvent) {
15     profile = loadProfileEvent.getCurrentTarget();
16     barceloPersonalization();
17   }
18   void loadAccomodationHandler(object sender, LoadAccommodationEvent loadAccomodationEvent) {
19     accommodations.Add(loadAccomodationEvent.getCurrentTarget());
20     barceloPersonalization();
21   }
22   void barceloPersonalization() {
23     foreach(IAccommodation accommodation in accommodations) {
24       if (!done && profile != null &&
25         profile.RegistrationType.Equals("Passport") &&
26         accommodation.Name.Equals("Barceló Costa Vasca")) {
27         AddViewModAccommodationEvent ev =
28               new AddViewModAccommodationEvent(accommodation, "<td class=\"text\"><a href=\"\
29         Accommodation.OnSignal(ev); done = true;
30     }}
31   }}
```

Fig. 3. *Mods* as plugins that import Modding Interfaces (line 8)

places it as appropriate. The only aspect known in advance is the type of the HTML fragment as indicated in the event payload when annotating the Modding Concepts.

4.2 Impact on Partners: Defining Mods

Unlike the open-source approach, OP restricts code access through the Modding Interfaces. Mod expressiveness is that of monotonic additions to the content of the host. Deletions are not permitted. Implementation wise, this means *mods* can extend the content of existing Views, and add new Views & Controllers.

Extending Existing Views. The programming model for *mods* is event-based. First, a mod subscribes to publishing events to collect data about the User Model and the Domain Model that is going to be rendered. Second, a mod signals processing events to indicate the availability of an HTML fragment ready to be injected in the current View. Therefore, the mod is totally unaware of all, the Model classes, the Controllers and the Web Forms that are in operation. From the *mod* perspective, the website is wrapped as a set of *Modding Concepts* and their corresponding events. Figure 3 shows the *mod* to be provided by the hotel partner for the rule: *"a 50% discount on room booking over the weekend is offered, provided the attendee holds a full passport":*

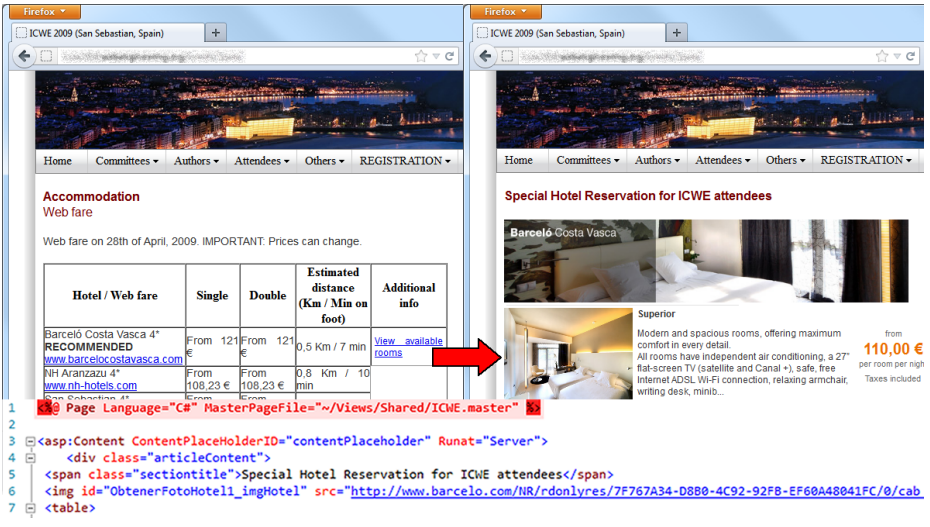

Fig. 4. A *mod* that introduces a new View & Controller. In the up side, the host's View links to the partner's View and the rendering of the partner's View. In the down side the partner's View code refers to the host template (i.e. MasterPageFile).

- a *mod* works upon Modding Concepts (e.g. *Accommodation* and *Profile*). This implies obtaining the classes for the corresponding interfaces (e.g. *IModdingConceptAccommodation* and *IModdingConceptProfile,* line 6). These classes' instances are obtained dynamically using dependency injection (see next section). This explains the *[ImportingConstructor]* annotation.
- a *mod* can subscribe to Publishing Events (e.g. *LoadProfile*, *LoadAccommodation*). This entails associating a handler to each Publishing Event of interest (lines 11, 12).
- a *mod* can signal Processing Events (e.g. *AddViewModAccommodation*). This signal is enacted in the context of a personalization rule. This rule is just a method (e.g. *barceloPersonalization*) which proceeds along three stages: (1) checks the pertinent aspects of the User Model and Domain Model as obtained from the Publishing Events (e.g. variables "*profile*" and "*accommodation*"); (2) constructs the event payload (i.e. an HTML fragment) and creates the event at hand; and finally (3), signals the Processing Event.

Adding New Views and Controllers. In the previous example, the output of the *mod* could have contained links to Views with additional information (e.g. room pictures). Figure 4 provides an example. These Views are kept as part of the ICWE website but they are provided by the partners. This requires the partner not only to extend host Views with "hooks" (i.e. a link to the partner View), but also to facilitate his own View and Controller. Partners' Controllers are like host Controllers. Partners' Views are like any other View except that they refer to the (rendering) template of the host so that the look&feel and non-contextual links of the hosting site are preserved (see Figure 4). This permits the partner's Views to link back to the rest of the website.

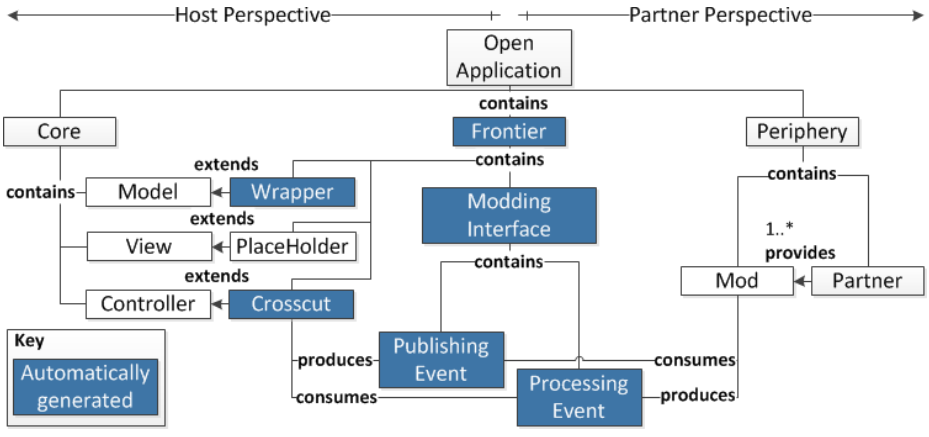

Fig. 5. Decoupling the *Core* from the *Periphery*: a model of the involved concepts

5 Open Personalization: Architecture

This section introduces the main architectural elements that ground the semantics of the *[ModdingConcept]* annotation. That is, the artefacts and associations to be generated as a result of a Domain Concept being turned into a Modding Concept. *Specifically, each annotation automatically outputs the following types of artefacts: Wrappers, Crosscuts and Modding Interfaces.*

Figure 5 outlines the main artefacts and conceptual relationships of our architecture. An **Open Application** contains a **Core**, a **Frontier** and a **Periphery.** The Core stands for the traditional architecture along the Model-View-Controller pattern. The Periphery includes the **Mods** provided by the **Partners**. Finally, the Frontier mediates between the Core and the Periphery through **Modding Interfaces**. Modding Interfaces encapsulates Model classes through events. **Publishing Events** are *<consumed>* by the Mods but *<produced>* by the Core. Alternatively, **Processing Events** are *<produced>* by Mods but *<consumed>* by the Core.

Mods impact on the Core. This impact is supported by different means depending on the nature of the artefact at hand. For Model class, the impact is in terms of a **Wrapper**: a class that becomes a Modding Concept is encapsulated so that only modding properties can become event payloads. For Controller classes, the impact is supported as a **Crosscut** for each of the class methods. Each method handles a Web Form (i.e. denoted in the code as *"return View(webFormName)"*). The Crosscut is "an aspect" that extends the base method with an "after advice" with two duties: (1) raising a Publishing event for each concept instance to be loaded by the Web Form (e.g. hotel Barceló), and (2), handling the Processing Events raised by the *mods*. Finally, the View (i.e. the Web Forms) requires the introduction of **PlaceHolders** where the mod output is to be injected.

So far, the description seems to suggest that the Core knows in advance the mods to be instantiated. However, this is not the case: mods can be added at anytime. This implies hot deployment, i.e. the ability of adding new *mods* to a running Web server without causing any downtime or without restarting the server. The Core cannot have an explicit dependency on *mods*. Inversion of Control and Dependency Injection are two related ways to break apart dependencies in your applications [6]. Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work. Instead, they get the objects that they need from an outside source. Dependency Injection (DI) means that this is done without the object intervention, usually by the "assembler" that passes constructor parameters and set properties. The assembler is a lightweight object that assembles different components in the system, in order to produce a cohesive and useful service.

In our architecture, Controllers are the component in charge of instantiating the *mods*. However, these instantiation are not achieved directly by the Controllers but through an assembler. That is, Controllers become IoC compliant components (a.k.a. parts), i.e. they do not go off and get other components that they need in order to do their job. Instead, a Controller declares these dependencies, and the assembler supplies them. Hence, the name Hollywood Principle: *"do not call us, we will call you"*. The control of the dependencies for a given Controller is inverted. It is no longer the Controller itself that establishes its own dependencies on the mods, but the assembler.

6 Revising the OP Requirements

Resilience. *Mods* should be resilient to View upgrades. This is the rationale of the Modding Interface: changes in the content or layout of a View should not impact the *mod*. Even if a concept (e.g. *Accommodation*) is no longer rendered, the mod will still raise the event, but no View will care for it. No dangling references come up. The mod becomes redundant but not faulty. And vice versa, new Views can be introduced where *Accommodation* data is rendered. This has no impact in the *mod*. Just the payload of the signalled event (i.e. the HTML fragment) will now start being injected in the place holder of the new View. This place holder should accept HTML fragments of the type being outputted by the *mod*. Otherwise, some disruption might occur that might eventually impact the rendering.

Extensibility. *Mods* can dynamically be added/deleted as partnership agreements change. Existing Model classes left outside partner agreements in the first round, might become Modding Concepts by just adding the corresponding annotations. However, this will require stopping the website to update the annotations and re-compile the code. This also raises the need for authorization mechanism so that not all partners will have access to all modding events. Grant and revoke privileges would be issued by the owner based on agreements with his partners. This is not yet available.

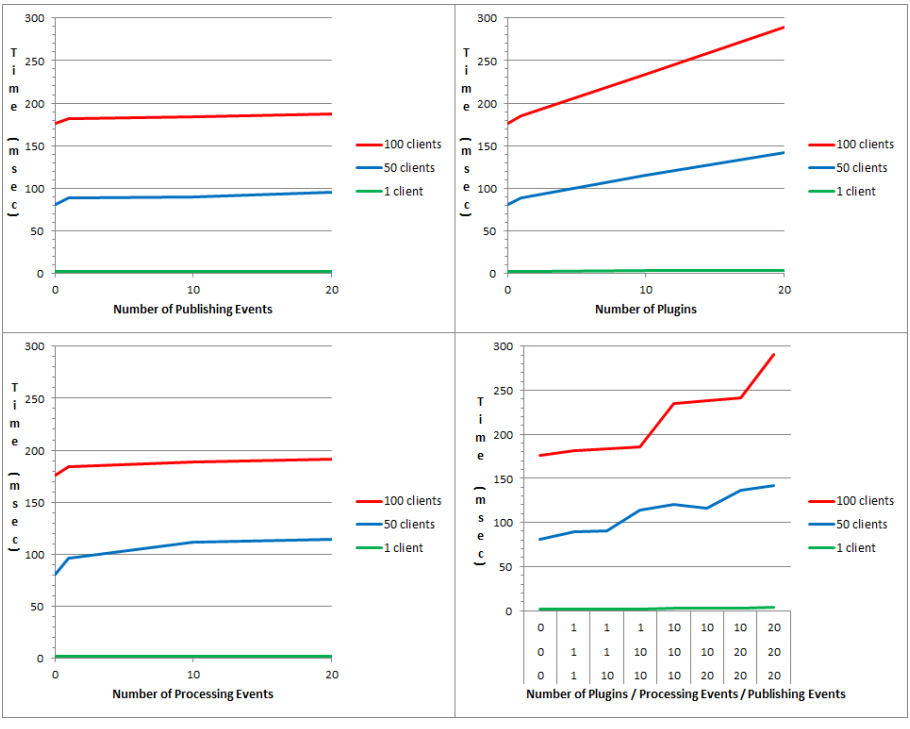

Fig. 6. Latency introduced by distinct OP factors (clockwise from bottom left): #Processing Events, #Publishing Events, #Plugins, and finally, the combined effect of all three

Scalability. *Mods* should not deteriorate the site performance. OP rests on a flexible architecture where (1) mods are installed dynamically and (2), mods interact with the Core through events. Both mechanisms trade flexibility for efficiency. Specifically, satisfying a URL request for a particular page now requires four additional steps: (1) instantiating the mod plugins at hand, (2) generating a publishing event for each Modding Concept in this page, (3) issuing a processing event for each mod that wants to contribute, and (4), capturing such processing events by the Controller at hand. As a general rule, end users should not pay a performance penalty for mods that are installed but not used during the current request. This section describes the results of a stress testing (a.k.a. load testing) of the OP architecture. The study evaluates the additional latency introduced when the ICWE site becomes mod-aware.

Stress testing entails a process of creating a demand on service, and measuring its response. Specifically, we measure the service that outputs the "Accommodation" page. The ICWE application has been deployed in an IIS 7.0 on Intel Core2 Duo T7500 2.2 GHz CPU with 4GB of memory. The test is conducted through Microsoft Web Capacity Analysis Tool (WCAT), a free lightweight HTTP load generation tool which is used to test performance

and scalability of IIS and ASP.NET applications [7]. WCAT is configured as follows: 30 seconds to *warmup* (no data collection)[2], 120 seconds of *duration* of simultaneous requests, 10 seconds to *cooldown* (no data collection), range of {1, 50, 100} *virtual clients* (concurrent clients over the same page), and finally, *request* stands for the petition of the *"Accommodation"* page.

The experiment is parameterized along the number of *mods*, the number of publishing event occurrences and the number of processing event occurrences for the request at hand. Figure 6 depicts the "time to last byte" metric for these three factors. For the ICWE-with-*no*-modding, the "Accommodation" request accounts for 2 msec. On top of it, OP introduces some affordable overheads. As suggested by the bottom right chart about the combined effect of the three factors, the event-based mechanism has minimal impact (i.e. the plateau in the charts stands for increases in the *#events* but keeping the *#plugins* constant). By contrast, the *#plugins* reveals itself as the factor with larger impact. Along the lines of IoC, each request implies to instantiate the involved plugins for the Controller at hand. For a hundred simultaneous requests, the impact of 1, 10, 20 plugins account for an increase of 5%, 33% and 64%, respectively. To be perfectly honest, we seldom envisage a scenario where a page is subject to over 20 plugins. We do not foresee more than 3/4 plugins per page on average, and this would represent a 15% penalty. Notice, that this number is just for satisfying the request, not to be confused to the elapsed time that the end user experiments. If normalized with the elapsed time (typically around 1300 msec.), the OP architecture represents around a 2% of increment for the most common envisaged scenarios.

Affordability. *Mods* should be easy to develop and maintain. *Mods* follow an event-driven style of programming. That is, the logic is split between event handlers and event producers. This is particularly helpful in our context where these event roles can be naturally split between partners and owners: partners focus on what should be the custom reaction (i.e. processing events) for the rendering of Modding Concepts, while owners focus on signalling when Modding Concepts are displayed (i.e. the Publishing events). This certainly leads to cleaner code. On the downside, the flow of the program is usually less obvious.

7 Related Work

Extensible architectures are a long-standing aim in software [11,5]. As a first requirement for in-house development, extensibility is becoming a must to integrate code from third parties. The motivation here is *"to integrate and build on existing work, by writing only the specialized code that differentiates them from their competition"* [1]. The ability to respond quickly to rapid changes

[2] WCAT uses a "warm-up" period in order to allow the Web Server to achieve steady state before taking measurements of throughput, response time and performance counters. For instance there is a slight delay on first request on ASP.NET sites when Just-In-Time (JIT) compilation is performed.

in requirements, upgradeability, and support for integrating other vendors' components at any time, all create an additional push for flexible and extensible applications, and grounds the work of Web architectures such as PLUX .NET [9], that resembles MEF, the .NET library we utilize to support OP. In the Java realm, the Open Services Gateway Initiative (OSGI) [16] framework propose a dynamic component model for Java, i.e. a way for components (known as bundles) to be started, stopped, updated and uninstalled without the need to reboot the system. OSGI also includes a way to define dependencies between bundles but it does not preclude any communication mechanism between components. Compared with an OSGI-like architecture, our approach rests on a "core component" (i.e. the website) and a set of "satellite components" where the interaction is only permitted from the core to the satellites. From this perspective, our approach is more rigid but it reflects the asymmetric relationship between the website owner and the third parties.

More akin with the OP vision is SAFE [13] an architecture of Web Application extensibility aimed at permitting users to personalize websites. SAFE is based on a hierarchical programming model based on f-units (the component model). An f-unit clusters all code fragments for a specific functionality within a web page, including the business logic, the visual appearance, and the interaction with users or other f-units. A web page is modelled as a so-called "activation tree" in which f-units are organized hierarchically, and activation flows top-down (naturally corresponding to the hierarchical DOM structure of an HTML page). Thus, a user who would like to personalize an application simply has to replace an existing f-unit with a new f-unit of her choice. Such customizations are dynamic in that f-units are registered and activated without stopping the running system. F-units contain SQL statements and this serves to support an implicit interaction between f-units sharing the same data. The bottom line is that SAFE proposes a more innovative mean for open participation by introducing a hierarchical model to web programming. This is simultaneously the main benefit, but also jeopardy, of SAFE. By contrast, we advocate for a more *evolutionary* approach. OP only makes the assumption of the MVC pattern and code annotation, and uses the well-known event-based programming model as the interaction mechanism. Capitalizing on existing techniques and programming models will certainly facilitate partner participation. The challenge is not only on *pluggable* components/f-units/mods but also *affordable*, *risk-controlled* technology that facilitates partner engagement. We use an existing technology (*.NET*) and use annotations to leverage from the general-purpose technology to domain-specific concepts. This motivates the conceptual leveragement of Model Classes into Modding Concepts. The notion of Modding Concept attempts to reduce the conceptual gap for partners and owners to understand each other while maximising decoupling by using events as the interaction mean.

8 Conclusions

Fostering a win-win relationship between website owners and partners, substantiates the efforts of Open Personalization (OP). This paper's goal was

to demonstrate that OP is feasible with existing technologies such as *.NET*. Though proving feasibility requires focusing on a specific platform, the approach is easily generalizable to any framework that supports "Inversion of Control". As future development, we plan to look into ways for partners to extend the User Model (i.e. the profile base). The profile base as designed by the host, might be insufficient to conduct some *mods*. Permitting partners to seamless define and collect additional user information through the website is certainly a challenge. Besides the technical challenges, OP also introduces new business models that need to be investigated.

Acknowledgements. This work is co-supported by the Spanish Ministry of Education, and the European Social Fund under contract TIN2011-23839 *(Scriptongue)*.

References

1. Birsan, D.: On Plug-ins and Extensible Architectures. ACM Queue 3, 40–46 (2005)
2. Bloomberg, J.: Events vs. services. ZapThink white paper (2004)
3. Brusilovsky, P.: Methods and Techniques of Adaptive Hypermedia. User Modeling and User Adapted Interaction 6, 87–129 (1996)
4. Cingil, I., Dogac, A., Azgin, A.: A Broader Approach to Personalization. Communications of the ACM 43, 136–141 (2000)
5. Erl, T.: A Comparison of Goals - Increased Extensibility. In: SOA Principles of Service Design, p. 451. Prentice Hall (2007)
6. Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern (January 2004), `http://martinfowler.com/articles/injection.html`
7. Friedman, E.M., Rosenberg, J.L.: Web Load Testing Made Easy: Testing with WCAT and WAST for Windows Applications. In: Proceesings of the 29th International CMG Conference, Dallas, Texas, USA, pp. 57–82 (December 2003)
8. Hippel, E.V.: Open source software projects as user innovation networks. In: Proceedings of the Open Source Software: Economics, Law and Policy, Toulouse, France (June 2002)
9. Jahn, M., Wolfinger, R., Mössenböck, H.: Extending Web Applications with Client and Server Plug-ins. In: Software Engineering, pp. 33–44 (2010)
10. JCP: JSR 168: Portlet Specification Version 1.0 (2003), `http://www.jcp.org/en/jsr/detail?id=168`
11. Oberndorf, P.: Community-wide Reuse of Architectural Assets. In: Software Architecture in Practice. Addison-Wesley (1997)
12. O'Reilly, T.: The Architecture of Participation (June 2004), `http://oreilly.com/pub/a/oreilly/tim/articles/architecture_of_participation.html`
13. Reischuk, R.M., Backes, M., Gehrke, J.: SAFE Extensibility for Data-Driven Web Applications. In: Proceedings of the 21st World Wide Web Conference, Lyon, France, pp. 799–808 (April 2012)
14. Riepula, M.: Sharing Source Code with Clients: A Hybrid Business and Development Model. IEEE Software 28, 36–41 (2011)
15. Robie, J., Hors, A.L., Nicol, G., Hégaret, P.L., Champion, M., Wood, L., Byrne, S.: Document Object Model (DOM) Level 2 Core Specification. Tech. rep., W3C (2000)
16. The OSGi Alliance: OSGi Service Platform Core Specification, Release 4.3 (2011)