Open Personalization: Involving Third Parties in Improving the User Experience of Websites

Dissertation
presented to
the Department of Computer Languages and Systems of
the University of the Basque Country in
Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
("*international*" mention)

Cristóbal Arellano Bartolomé

Supervisors:
Prof. Dr. Oscar Díaz García
Dr. Jon Iturrioz Sánchez
San Sebastián, Spain, 2013

This work was hosted by the *University of the Basque Country (Faculty of Computer Sciences)*. The author enjoyed a doctoral grant under de **FPI** (***F**ormacion de **P**ersonal **I**nvestigador*) from the *Spanish Ministry of Science & Education* during the years 2007 to 2011. The work was was co-supported by the *Spanish Ministry of Education*, and the *European Social Fund* under contracts (TIN2005-05610), *MODELINE* (TIN2008-06507-C02-01) and *Scriptongue* (TIN2011-23839).

Summary

Traditional software development captures the user needs during the requirement analysis. The Web makes this endeavour even harder due to the difficulty to determine who these users are. In an attempt to tackle the heterogeneity of the user base, *Web* ***Personalization*** techniques are proposed to guide the users' experience. In addition, ***Open*** *Innovation* allows organisations to look beyond their internal resources to develop new products or improve existing processes.

This thesis sits in between by introducing ***Open Personalization*** as a means to incorporate actors other than webmasters in the personalization of web applications. The aim is to provide the technological basis that builds up a trusty environment for webmasters and companion actors to collaborate, i.e. *"an architecture of participation"*. Such architecture very much depends on these actors' profile. This work tackles three profiles (i.e. software partners, hobby programmers and end users), and proposes three *"architectures of participation"* tuned for each profile. Each architecture rests on different technologies: a *.NET* annotation library based on *Inversion of Control* for software partners, a *Modding Interface* in *JavaScript* for hobby programmers, and finally, a *domain specific language* for end-users. Proof-of-concept implementations are available for the three cases while a quantitative evaluation is conducted for the *domain specific language*.

Contents

1 Introduction **1**

1.1 Context 1
1.2 General Problem 2
1.3 This Dissertation 4
1.4 Contributions 6
1.5 Research Approach 7
1.6 Outline 9
1.7 Conclusions 10

2 Background **13**

2.1 Introduction 13
2.2 Web Personalization 13
2.2.1 Definition & Motivation 14
2.2.2 Engineering Adaptive and Adaptable Hypermedia Systems 16
2.2.3 Successful Case Studies 18
2.2.4 Current Research Issues 19
2.3 Web Augmentation 20
2.3.1 Definition & Motivation 20
2.3.2 Web Augmentation through an Example: The BookBurro Script 22
2.3.3 Successful Case Studies 23
2.3.4 Current Research Issues 25

2.4 Conclusions 26

3 Server-Side Open Personalization **27**

3.1 Introduction 27
3.2 Motivating Scenario and Research Question 28
3.3 Requirements 30
3.3.1 Existing Solutions 31
3.3.2 Our Contribution 33
3.4 The Modding Interface 34
3.5 Specification of the Modding Interface 36
3.6 Impact on the Host: Making a Website Mod-Aware 37
3.7 Impact on Partners: Defining Mods 39
3.8 Architecture 42
3.9 Discussion 44
3.9.1 Resilience 44
3.9.2 Extensibility 45
3.9.3 Scalability 45
3.9.4 Affordability 47
3.10 Conclusions 48

4 Hybrid Open Personalization **51**

4.1 Introduction 51
4.2 Motivating Scenario and Research Question 53
4.3 Requirements 55
4.3.1 Existing Solutions 56
4.3.2 Our contribution 58
4.4 Crowdsourcing Web Augmentation 58
4.5 The Modding Interface: a Client-Side Perspective 61
4.6 Specification of the Modding Interface 62
4.7 Script Development 65
4.8 Script Testing 67
4.9 Script Advertising 72

4.10 Script Sandboxing 76
4.11 Discussion 80
4.11.1 Affordability 80
4.11.2 Resilience 81
4.11.3 Scalability 82
4.11.4 Security 83
4.12 Conclusions 83

5 Client-Side Open Personalization 85
5.1 Introduction 85
5.2 Motivating Scenario and Research Question 87
5.3 Requirements 91
5.3.1 Existing Solutions 94
5.3.2 Our Contribution 100
5.4 Web Augmentation: Caring for Producers 101
5.4.1 Sticklets 106
5.4.2 StickletBox 109
5.4.3 The Issue of Entity Linkage 111
5.4.4 The Issue of XPath Complexity 118
5.4.5 The Issue of Non-HTML Sources 122
5.4.6 The Issue of Note Rendering 124
5.4.7 The Operational Semantics of Sticklets 126
5.5 Web Augmentation: Caring for Consumers 129
5.5.1 Trustworthiness 129
5.5.2 Maintainability 134
5.6 Web Augmentation: When Attention is Scarce 135
5.6.1 Greasemonkey Operation 135
5.6.2 No Time to Code 136
5.6.3 No Time to Install 139
5.6.4 No Time to Share 142
5.7 Evaluation 143
5.7.1 Sticklet consumption for computer literates 145

5.7.2 Sticklet production for hobby programmers 152
5.8 Discussion 157
5.8.1 Expressiveness 157
5.8.2 Learnability 158
5.8.3 Trustworthiness 158
5.8.4 Maintainability 159
5.8.5 Understandability 159
5.8.6 Tailorability 159
5.8.7 Operability 160
5.8.8 Provisionability 160
5.8.9 Installability & Shareability 160
5.9 Conclusions 161

6 Conclusions 163
6.1 Overview 163
6.2 Results 163
6.3 Publications 165
6.4 Research Stage 167
6.5 Assessment and Future Research 168
6.6 Conclusions 170

Bibliography 173

List of Figures

1.1 Collaborative Software Development diagram. 3
1.2 Longtail effect in software development. 4
1.3 The relation between user affordance and contribution value in *Open Personalization*. 5
1.4 Chapter map of the dissertation. 9

2.1 *Domain Model* and *User Model* of a bookshop in *Hera*. . . 15
2.2 *Navigation Model* of a bookshop in *Hera*. 16
2.3 The bookshop after and before of the adaptation. 18
2.4 *Augmented Reality* through an example. 20
2.5 *Amazon* before and after the *BookBurro* augmentation. . . 21
2.6 *BookBurro* in *JavaScript*. 24

3.1 Domain classes annotated to become *Modding Concepts*. . 38
3.2 Mod-aware View: the ASPX includes a placeholder for mod's output. 39
3.3 *Mods* as plugins that import *Modding Interfaces*. 40
3.4 A *mod* that introduces a new *View* & *Controller*. 41
3.5 Decoupling the *Core* from the *Periphery*: a model of the involved concepts. 42
3.6 Latency introduced by distinct SOP. 46

4.1 Raw page vs. Augmented page. 53
4.2 *DblpFigures* script: *DOM Events* vs. *Conceptual Events*. . 54

4.3 The Metropolis Model. 59
4.4 *ICWE Website Modding Interface*. 63
4.5 Testing *dblpFigures* mod through the *JsUnit* framework. . 70
4.6 Advertising user scripts through the web site. 73
4.7 Code that executes the preview mode in the client. 75
4.8 Augmentation at run-time: *DOM tree* evolution. 76
4.9 The *Modding-Interface* Architecture. 77
4.10 *Weaver*'s code: loading augmentation scripts. 79
4.11 Augmentation-enabled page:
meta-data about the *Modding Interface*. 80

5.1 *Amazon* before and after the *BookBurro* augmentation. . . 87
5.2 *BookBurro* in *JavaScript*. 90
5.3 *Sticklet* Design Drivers. 92
5.4 *BookBurro* using *Chickenfoot* API. 99
5.5 Feature diagram for the *Web Augmentation* domain. . . . 102
5.6 *BookBurro* as a *sticklet*. 104
5.7 *Sticklet*: abstract syntax. 105
5.8 *Sticklet* for augmenting *Amazon* with the book reservation
at the Manchester University Library. 110
5.9 Entity linkage through searching (code). 113
5.10 Entity linkage through searching (views). 115
5.11 Entity linkage through mapping (views). 116
5.12 Entity linkage through mapping (code). 117
5.13 The *assisted* mode. 119
5.14 *Note* rendering.
Default rendering supplemented with *HTML* directives. . . 123
5.15 The operational semantics of *Sticklet*. 125
5.16 *Sticklet* tracing exemplified for *BookBurro*. 131
5.17 *Sticklet* error reporting. 133
5.18 *Sticklet* inline editor. 138
5.19 Installation of a *sticklet* from a tweet. 140

5.20 *Sticklet* inline sharing. 142

Chapter 1

Introduction

1.1 Context

Enhancing requirement elicitation is a long-lasting endeavour of software development. The existence of conferences, journals and seminars around this issue almost since the inception of software, evidences both the importance and the difficulties of this challenge. The Web2.0 movement is shading a new light on this topic. The notion of "the perpetual beta" to denote a *continuum* in the improvement of applications, or "open innovation" as a way to enlarge the number of both stakeholders and developers, offer new means to an old problem. These notions are usually supported by an *architecture of participation*, namely, by systems that are designed for user contribution [O'R04]. This thesis focuses on web applications, and looks into inclusive approaches to web development that facilitate the participation of different actors in application coding.

Traditional software development captures the user needs during the requirement analysis phase. In this phase, software analyst tries to capture

all the user (a.k.a. stakeholder) needs. Because it is very difficult to capture all user requirements, and implement all of them is expensive, only a subset of them is finally supported. As a result, software that satisfies part of these needs is created and made available to the user. This fact has two implications. First, only a subset of the stakeholders' requirements is satisfied. Second, the adaptation of such system to new requirements normally is under the control of IT department. Additionally, companies do not work in isolation but interacts with other companies known as partners. These relationships can be as important as the main activity of the company because sometimes it is difficult or impossible to perform the company main activity without them. The nature of relationships between business partners is win-win and must be taken into account when creating an application.

In a web scenario, it is more difficult to determine who the users are. The number of potential web users and then the required functionality to fulfil their needs increases. *Web Personalization* aims to cluster types of users in roles and adapt the given information to such roles [CDA00]. A way to let users satisfy their needs is desirable. User innovation and customer centric development are two topics that are changing the way the software is conceived. The user is no longer a passive actor during the analysis phase but voluntarily contributes during the rest of the software development phases like implementation or testing.

1.2 General Problem

So far, most of the software development is realized inside a company (intra-company). This means that software development is realized in isolation. In the best case, partners are engineered into the application as a user role. They do not have any chance to reflect its own activity. This fact hinders partners business activity and hence the activity of the company. It is needed a way to search synergies and allow third parties to contribute to the company application (inter-company) to foster a win-win relationship

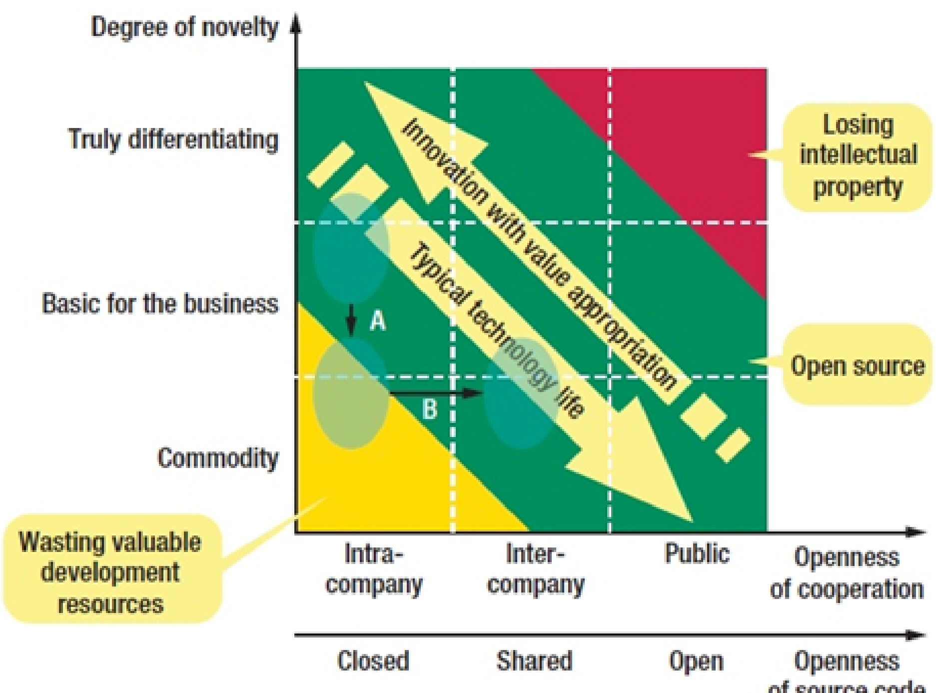

Figure 1.1: Collaborative Software Development. A diagram that helps to choose the openness of a solution based on its novelty.

between them. Figure 1.1 illustrates this scenario [Rie11]. The x-axis indicates the openness in cooperation whereas the y-axis represents the degree of novelty. The two coloured corners must to be avoided, the yellowish because it incurs in a waste of time and money when developing low-valued software and the reddish because the company loses the intellectual property of distinctive software. Traditionally, the software development occurs inside a company (left column). At the beginning of the development, its novelty is high but as time goes its novelty decays (transition a). When a company spends its resources developing commodity software, it should examine the possibility of open it to their partners (transition b). *Web Personalization* belongs to this type of software hence it can be developed in a cost-effective way when developed with the collaboration of partners.

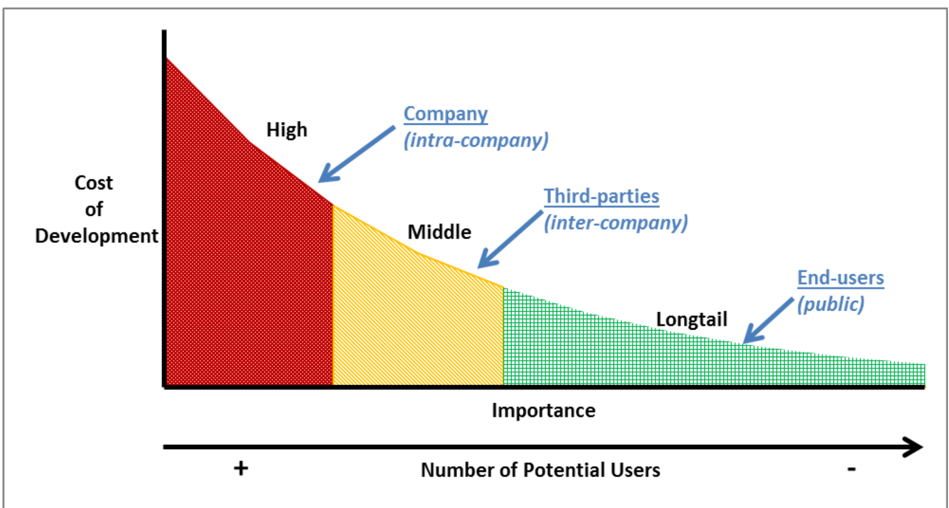

Figure 1.2: The Longtail effect applied to software development.

Additionally, it is not possible to foresee all the requirements of the user. In the best case, one-size-fits-all solutions are replaced by role-personalized solutions. The functionality that is going to be reflected into the application is the functionality that covers most of the users. It means that the functionality required by very few people is not going to be taken into account. The "voice of the client (VOC)" is not heard. It is needed a way to allow clients to satisfy the requirements not satisfied by the company, possibly without company help (public).

1.3 This Dissertation

This dissertation faces the previous problems depending on the importance of the functionality required. Figure 1.2 depicts the three main areas. The first area represents **highly-requested functionality**. It is the main/core functionality offered to the customers and it is provided by the company itself (intra-company). The second area represents **moderately-requested functionality**. Such functionality, being relevant, is not provided by the company itself but by third parties (inter-company). So here, two

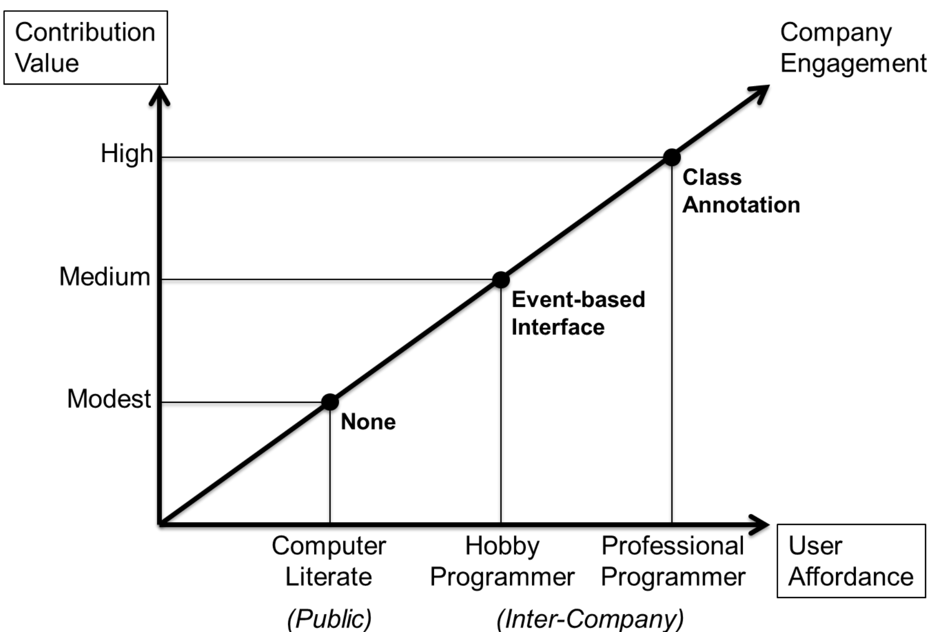

Figure 1.3: The relation between user affordance and contribution value. Greater contributions requires more user engagement and complex mechanisms.

architectures of participation are proposed to foster a win-win relationship between the company and the third parties. Last but not least, the longtail area denotes **modestly-requested functionality**. The poor payoff of this functionality makes it fall outside the company's radar. However, it can be paramount for some few users (public). Here, the approach is to empower end users for them to develop such functionality by themselves.

This thesis addresses the following research question:

> ***How can actors be empowered to satisfy their own requirements for web applications?***

Specifically, we look at three different actors: partners, hobby programmers (inter-company), and computer literates (public). Each actor profile brings his own skills and limitations, and hence, conditions the solution. Figure 1.3 illustrates this tension between user affordance, contribution value and the engagement required by the owner of the

website. When company engagement is none, computer literate as low-skilled actor, can make modest contributions. When company offers a greater engagement based on event-based interfaces, hobby programmer is the required profile that can make medium contributions on top of such interfaces. Finally, when company engagement is even greater and offers annotated classes, professional programmer (partner) profile can make greater contributions based on such annotations. Next section enumerates the main contributions of this dissertation.

1.4 Contributions

In this dissertation the following three scenarios are tackled:

Empowering Partners

- Problem statement: A partner is a software company that develops on top of a third-party application. How can partners extend third-party applications in a safe, reliable and affordable way?

- Solution: We propose "an architecture of participation". This architecture lets partners extend company web application with their own code in a reliable way. The company defines an interface for partners and let them to build on top of it. *Class annotations* and *Inversion of Control* are the main technical approaches.

Empowering Hobby Programmers

- Problem statement: Hobby programmers are individuals that have a strong programming background but use this skill outside the work realm. In the web community, the *userscript* community showcases this situation. ~~Those scripts enhance web applications in different ways. *Userscripts* is a techy community where final users of the enhanced applications might never hear about these enhancements.~~

How can the web application owners tap into this community for the good of both the hobby programmers and its own final users?

- Solution: An architecture of participation is proposed. This architecture lets scripters extend company application with their own contributions. The company defines an event-based interface for scripters and let them build on top of it using *JavaScript*. ~~So provided scripts might be readily shared through the website directly rather than through a script repository.~~

Empowering Computer Literates

- Problem statement: Computer literacy is defined as the knowledge and ability to use computers and related technology efficiently. As digital natives grow up, most people will become computer literates. How can end users be empowered to tune the existing websites to their own needs without resorting to the IS department?

- Solution: We investigate on the use of domain specific languages for end users. The aim is to abstract technical details into programming metaphors that facilitate end user implication. Excel formulae language is a successful example for spreadsheats. Here, we look at the web realms using client-based augmentation techniques.

1.5 Research Approach

Two research approaches dominate the research in Information Systems: behavioural science and design science [HMPR04]. The former is applied to search the reasons underlying the human behaviour, and its output is a theory that gathers them. The latter is used to extend the capabilities of human behaviour and its output is an artefact (constructs, models, methods and instantiations as proposed by [MS95]) that helps to achieve this aim. This thesis follows the design-science approach; this work enables third

parties and end users to extend the web and the output is a set of software architectures. The research guidelines of design-science proposed by [HMPR04] are applied to this thesis as follows:

- *Design as an artefact & Research contributions*. As an output of this thesis, a set of models and their instantiations are proposed. These outputs are described in depth in the Section "Our contribution" of Chapters 3, 4 and 5.

- *Problem relevance*. The problem in general and its decomposition in particular are stated in this chapter, in Sections 1.2 and 1.4. In Chapters 3, 4 and 5, the Section "Motivation" describes the context and its relevance, and the problem associated to such context.

- *Design evaluation*. The architectures are illustrated using working examples. Additionally, in Chapter 5, a quantitative and qualitative evaluation was performed to assess the usability of the proposed solution.

- *Research rigor*. The proposed solutions make use of standard notations when available. In the cases where there is an absence of standards, the most widespread alternative has been selected. In addition, all of the contributions of this thesis are based on the state of the art and compared with other works.

- *Design as a search process*. All the contributions mentioned in this thesis are the result of the searching a solution to the problems introduced in this chapter. This process begins with the search of related work that solves similar problems and is described in the Chapters 3, 4 and 5, concretely in the "Existing Solutions" Sections. Taking into account other works and detecting their weaknesses in our context, a new solution is provided.

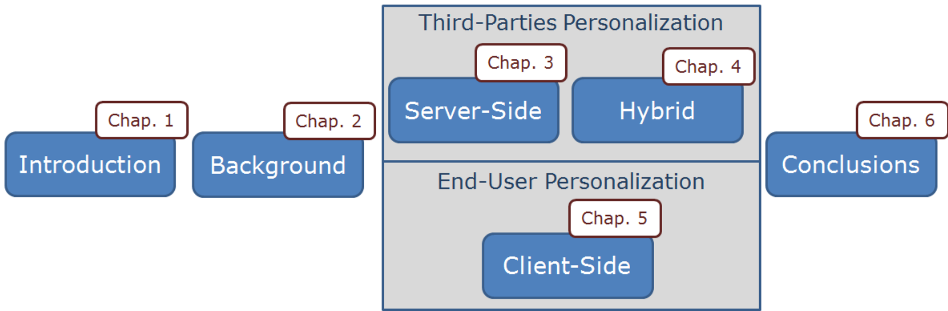

Figure 1.4: Chapter map of the dissertation.

- *Communication of the research.* The research performed in this thesis was communicated to the audience through academic conferences and journals of the *Web Engineering* area listed in Chapter 6, in Section 6.3. This thesis complements the previous communications, putting all the findings in the same context.

1.6 Outline

The content of each chapter is summarized in this section. Figure 1.4 contains a map that illustrates the relationships between the chapters of this dissertation.

Chapter 2

This chapter presents two topics that explain the adaptation of web content in order to satisfy the user needs, namely *Web Personalization* and *Web Augmentation*. These topics set the bases for the rest of this thesis.

Chapter 3

This chapter presents a scenario where some of the personalization effort of the company's web application is delegated to the company partners. To support such scenario, an architecture of participation is proposed. This architecture is based on an interface (i.e. *Modding Interface*). Using

the *Modding Interface*, the company specifies what can be externally personalized, while company partners can build personalizations that are going to be published through the company's web application.

Chapter 4

In this chapter, we envision an architecture of participation between the company and its customers. The proposed architecture permits the company specify what can be personalized by scripters using an event-based interface. In return, the scripters might be allowed to provide their contribution as part of the content of the website.

Chapter 5

This chapter presents an approach for end-user client-based customization of websites. To this end, a *domain specific language* is introduced: *Sticklet*. *Sticklet* allows end users to create, understand, maintain and share its own extensions.

Chapter 6

This chapter concludes the dissertation by remarking the main results, listing publications of the author's thesis, enumerating the limitations of the current solutions and proposing future work.

1.7 Conclusions

In this chapter, *Open Personalization* as a means to incorporate actors other than webmaster in the personalization of web applications is introduced as the main topic of this dissertation. For such topic, three scenarios are identified based on the profile of the actors; partners, hobby programmer and end users. For each scenario, the related contributions are listed: the

first based on code annotations for partners, the second based on event-based interfaces for hobby programmers and the third based on a domain specific language for end users. Finally, the design science as research approach is presented and it is explained how its guidelines are followed in this thesis. The next chapter provides the necessary background to understand the rest of the chapters.

Chapter 2

Background

2.1 Introduction

Web Personalization and *Web Augmentation* faces the adaptation of the content to the user needs. Whereas in *Web Personalization*, the adaptation is designed by the webmaster; in *Web Augmentation* the adaptation is performed at the back of the webmaster, possibly by the users of the website.

This chapter introduces *Web Personalization* and *Web Augmentation,* establishing in this way the basis of the rest of the thesis. For each area, the definition, the motivation of its existence, successful case studies and the current research issues are presented.

2.2 Web Personalization

Web Personalization refers to making a web site more responsive to the unique and individual needs of each user [CDA00]. To achieve this goal,

the web application is adapted to the user needs; the webmaster designs a website where its content/layout/navigation changes depending on the user.

In this section we do not want to emphasize the role played by the user of the system (e.g. differences between adaptive and adaptable) but in the extra burden of the webmaster to create a personalizable system (e.g. the creation/population/management of the *Adaptation Model* and *User Model*).

2.2.1 Definition & Motivation

The widely used definition of *Adaptive/Adaptable Hypermedia Systems* is:

"By adaptive hypermedia systems we mean all hypertext and hypermedia systems which reflect some features of the user in the user model and apply this model to adapt various visible aspects of the system to the user" [Bru96].

The previous definition can be decomposed in the following concepts:

- *Applicable to all hypertext and hypermedia systems.* Hypertext/Hypermedia systems are the subject to be adapted. According to Ted Nelson, hypertext is a collection of documents containing cross-references or "links" which, with the aid of an interactive browser program, allow the reader to move easily from one document to another.

- *Features of the user reflected in a user model.* Features of a user are put together in a user model. According to Kobsa, these features can come from user data provided directly by the user, usage data as a result of the user interacting with the hypertext and environment data that is information about the user location and platform [KKP01].

- *Adaptation based on user model.* Adaptation or changes in the hypertext model (documents and/or links) are based on the previous user model.

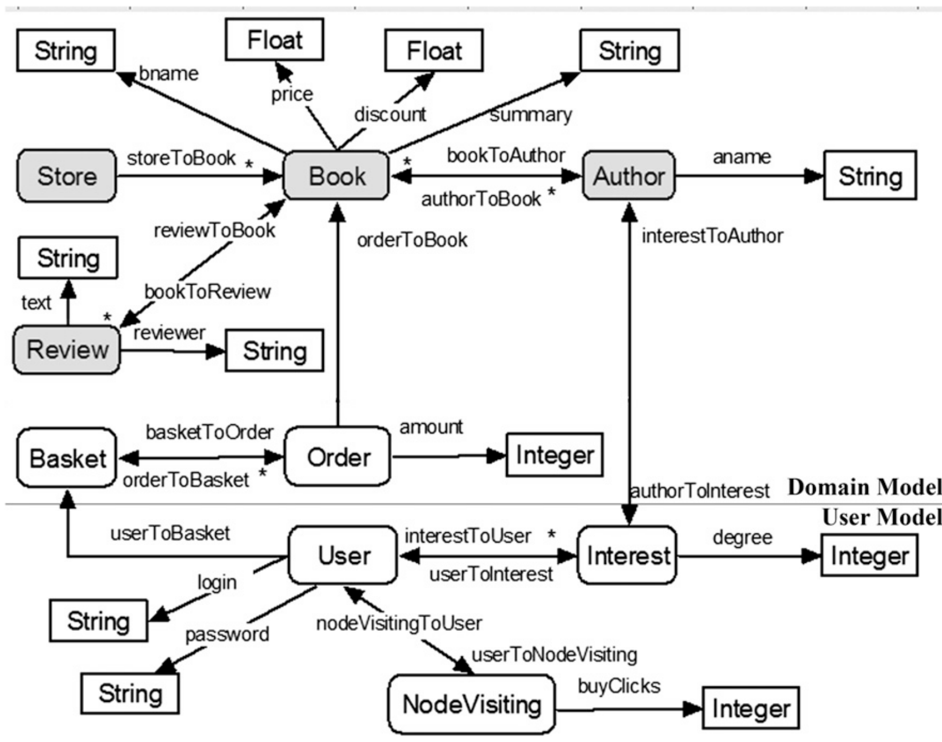

Figure 2.1: *Domain Model* and *User Model* of a bookshop in *Hera* design method.

The problem that *Adaptive/Adaptable Hypermedia* faces is the “lost in hyperspace” syndrome, which is described as "the user not having a clear conception of the relationships within the system or knowing his present location in the system relative to the display structure and finding it difficult to decide where to look next within the system" [EH98, EW85]. Such situation occurs when there is an information overload in hypermedia systems, there are normally too many content/links and the user has little knowledge how to proceed and select the best for him. *Adaptive/Adaptable Hypermedia Systems* tries to face this problem, reacting to the used needs by modifying the system itself. The following subsection illustrates web personalization using the adaptation of a bookshop as an example.

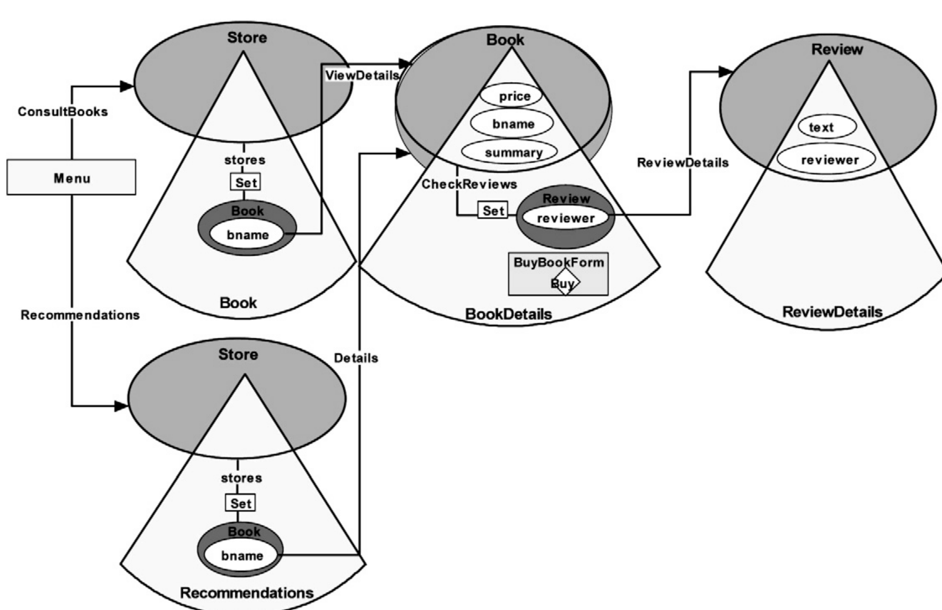

Figure 2.2: *Navigation Model* of a bookshop in *Hera* design method.

2.2.2 Web Personalization through an Example: The Bookshop

As an example, consider a bookshop. All the information about the example was extracted from [GGH10]. Web design methods define three main models: the *Domain Model*, which contains the structure of the domain data; the *Navigation Model*, that contains the structure and behaviour of the navigation view over the domain data, and the *Presentation Model*, in which the layout of the hypermedia presentation is defined. Figure 2.1 (top) shows the *Domain Model* of the bookshop with its main concepts: *Store*, *Book*, *Author*, *Review*, *Order* and *Basket*. Figure 2.2 depicts the *Navigation Model* of the bookshop that contains the links between the concepts of the *Domain Model*: *Details* link between *Store* and *Book, ReviewDetails* between *Book* and *Review,* etc.

According to the definition of *Web Personalization* previously introduced, engineering adaptive/adaptable hypermedia systems requires the addition of the *User Model* and the *Adaptation Model*. Both are briefly

described:

- *User Model.* This model captures the knowledge about the user. Concretely, it captures the *relevant knowledge* for the *current system.* The acquisition of such information is based on *user data*, *usage data* or *environmental data* [KKP01]. User data includes user explicitly provided data like preferences or skills. Usage data includes information about detected visited pages, link navigations or interactions inside a page. Environmental data gathers information about the user environment like geographical data or device screen size. Figure 2.1 (bottom) contains and example of *User Model* of the bookshop. This *User Model* stores user provided data (i.e. *Interest*) as well as usage data (i.e. *NodeVisiting*).

- *Adaptation Model.* This model captures the adaptation rules. These rules define how the *Domain Model* and *User Model* are used to adapt the current application. These rules follow the ECA (Event-Condition-Action) pattern, where events are triggered by changes in the models, and if the condition of the rule is met, the action reflects the updates in the corresponding model. Consider the following ECA rule in PRML format:

  ```
  When SessionStart do
   If ForAll (UM.User.usertoInterest)
    (UM.User.usertoInterest.degree < 100) then
    hideLink(NM.Recommendations)
   endIf
  endWhen
  ```

  Such rule is an adaptation rule that follows the ECA pattern. *When* a session starts (Event), *if* degree of user interest is below 100 (Condition), *hide* the link of recommendations (Action). In this example the rule queries the *User Model* (i.e. *UM*) and the adaptation performs changes over the *Navigation Model* (i.e. *NM*).

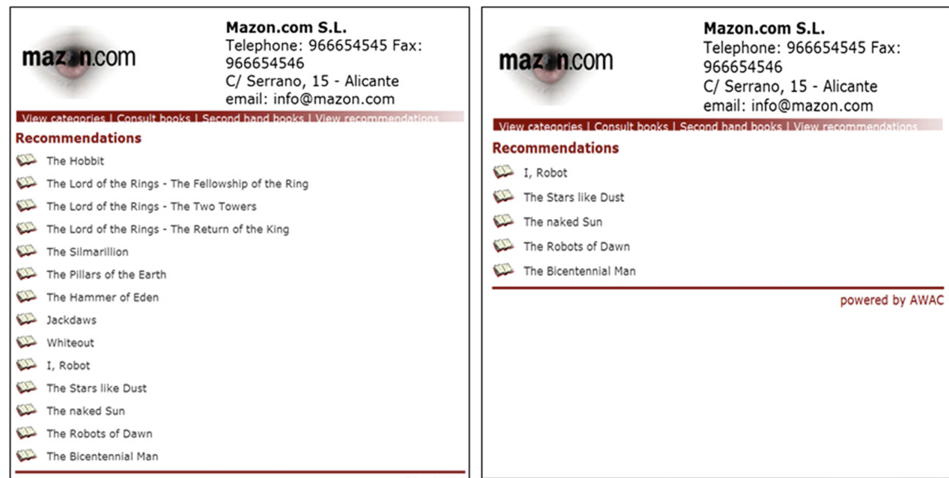

Figure 2.3: The bookshop after and before of the adaptation.

When these models are available, the system can use them to adapt to the user. This adaptation can affect the pages, as well as navigation [Bru01]. The adaptation of *page text presentation* affects to the addition/remove/sort/dimm of text information (e.g. text fragments) and the *page hypermedia presentation* affects to the selection for the quality/size of media (e. g. images or video). The adaptation of *navigation* affects to the links addition/remove/sort/hide/disable. Figure 2.3 shows the bookshop before (left) and after (right) the adaptation, in which the adaptation rule for the bookshop is applied over navigation by removing links (i.e. *hideLinks*) of the *Recommendations* page (i.e. *Recommendations*).

2.2.3 Successful Case Studies

The types of systems identified as be amenable to be personalized are six [Bru96]: *educational hypermedia systems, on-line information systems*, *online help systems*, *information retrieval hypermedia*, *institutional hypermedia* and *personalized views*.

In the academic research area, educational hypermedia is the area which predominates in the literature. The following works illustrates the

state of the art in such area: *ELM-ART* [BSW96], *ELM-ART II* [WS97], *KBS-Hyperbook* [HN99], *2L670* [BC98], *InterBook* [BE98] and *INSPIRE* [PGKM03]. In addition, generic platforms have been designed to face the adaptive web applications, no matter the domain as *OOHDM* [RSG01], *UWE* [Koc01], *Hera* [FH02], *OO-H* [GGC03], *WSDM* [CTB03] and *WebML* [DMP06].

From the commercial area, distinct 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).

2.2.4 Current Research Issues

Although *Web Personalization* area is mature, there are some topics such as user engagement; trust; user motivation, attention, and effort; recommender systems; user-centered design and evaluation; educational data mining; modeling learners; user models in microblogging; and visualization, in which researchers are still contributing [UMA12].

One of the topics that raises more interest is **user model interoperability** [CCG11, Mul12]. Nowadays there are a large number of user-adaptive systems and systems that collect personal information. The problem is that users interact with multiple systems that cannot exchange users' personal information (i.e. *User Models*). As a result, users' personal information tends to be sparsed and/or replicated in different systems and cannot be reused between them. This problem becomes worse when the website displays personalized content from its partners, because webmaster cannot personalize partner's content and partners cannot access the *User Model* of the website to perform the personalization.

Figure 2.4: *Augmented Reality* through an example. A image of London enriched with information about objects that appears in such image.

2.3 Web Augmentation

Web Augmentation is to the web what *Augmented Reality* is to the physical world: layering relevant content/layout/navigation over the existing web to customize the user experience. Figure 2.4 shows *Augmented Reality* in action, an image of London (reality) is enriched with information about the objects appearing in such image (augmentation). In *Web Augmentation* setting, augmentations are not made by the creator but by the user of the web application. *Web Augmentation* "take back the web" to the web users by allowing the customization of web applications by web users.

Being *Web Augmentation* an emerging area and trying to simplify the explanation, the concepts of this section are illustrated using an example.

2.3.1 Definition & Motivation

A widely used definition of *Web Augmentation* is:

Figure 2.5: *Amazon* before and after the *BookBurro* augmentation.

Web augmentation "adds content or controls not contained within the Web pages themselves to the effect of allowing structure to be added to the Web page directly or indirectly, or to navigate such structure" [Bou99].

Examples of what this technology generically enables include reorganizing page content, supplementing page data, changing fonts and formats, etc. [McF05, Fil06]. *Web Augmentation* is not new. Layering web code at the client-side on top of existing websites is being used to improve the affordance of third-party services. If this service is *Skype* calls, the augmentation plugin at [Sky05] turns any phone number found in a web page into a button that launches *Skype* to call that number. If this service is *AVG* security warnings, *LinkScanner* [AVG10] is an augmentation utility that permits to scan search results from *Google*, *Yahoo!* or *Bing*, and places a safety rating next to each recovered link. Besides in-place service invocation, *Web Augmentation* can support a broad range of situational scenarios:

- On browsing an online journal (*e.g.* *USA Today*), you can be interested in the coverage that a given headline receives in another online newspaper, e.g. *The NY Times*. Skipping to the *TNYT* and

searching for a related headline could be too cumbersome to do on a routine basis. Rather, you would like the *USA Today* website to be augmented with a button placed by the *USA Today* headline that directly pops up the summary at *TNYT* for this headline.

- When rendering a book at *Amazon*, it could be useful to know the prices/comments for this book at other online bookshops.

- On weighting a job post at *www.monster.com*, it could be of interest to supplement *monster* data with information about the range of wages and conditions of similar jobs as found in other web sites (e.g. *jobs.trovit.co.uk*).

Using special weavers, third-party *JavaScript* code can make on-the-fly changes to the currently loaded Web page. Weavers are available for *Firefox* (e.g. *Greasemonkey*), *Internet Explorer* (e.g. *IE7Pro* or *Turnabout*), *Safari* (e.g. *SIMBL + GreaseKit*), and natively supported in *Opera* and *Google Chrome*. The following subsection illustrates *Web Augmentation* using a popular example, *BookBurro* script for *Greasemonkey (GM)* [LBS05].

2.3.2 Web Augmentation through an Example: The BookBurro Script

As an example, consider a popular script: *BookBurro*[1]. This script embeds price comparison in *Amazon* pages. Concretely, when on an *Amazon* book page, a link is shown after the book ISBN. Clicking on such link, the *BookBurroPanel* is shown with a list of prices at other bookshops. Figure 2.5 shows the outcome before and after applying the script that injects the *BookBurroPanel*. This is achieved at the browser through the weaver. Weavers permit scripts to act upon web pages at runtime. Pages

[1] *BookBurro* is available at *http://userscripts.org/scripts/source/1859.user.js*

are realized as *DOM* trees[2]. The script is triggered by User Interface events (UI events) on this *DOM* tree (e.g. *load*, *click*). Event payloads provide the data to feed script handlers which, in turn, update the *DOM* tree. The script is outlined in Figure 2.6. The process goes as follows:

- interacting with a page triggers UI events (e.g. *load*),
- the script *reacts* to this event by triggering a handler (lines 6-39). The association between an event and a handler (a.k.a. event listener) is achieved through the *addEventListener* function (line 6),
- a handler can access any node of the page (using *DOM* functions such as *document.evaluate* in lines 9-10), and create *HTML* fragments (e.g. line 21),
- a handler can also *change* the *DOM* structure at wish by injecting *HTML* fragments (e.g. the *BookBurroPanel*). In the example, the output is injected at a point identified by an *XPath* expression on the underlying *DOM* structure (i.e. the injection point). *DOM* functions are used for this purpose (e.g. *appendChild* in line 23 and 36),
- this script is associated with a URL pattern that denotes the pages to which the script applies. This is specified through the @*include* annotation (line 3).

2.3.3 Successful Case Studies

Concrete examples can be found at *userscripts.org*, a popular repository for augmentation scripts. This site reports millions of downloads and

[2]The Document Object Model (*DOM*) is a platform- and language-independent standard object model for representing *HTML* or *XML* documents as well as an Application Programming Interface (API) for querying, traversing and manipulating such documents.

```
1  // ==UserScript==
2  // ...
3  // @include        http://www.amazon.com/*
4  // ...
5  // ==/UserScript==
6  window.addEventListener("load", function(){
7   // Script scope: structural condition + content condition on the current document
8   // Checking structural condition
9   var isbns=document.evaluate("//li[contains(b/text(),'ISBN-10')]",
10                               document, null, XPathResult.ANY_TYPE, null);
11  var nodes=[];
12  for(var elem=isbns.iterateNext();elem!=null;elem=isbns.iterateNext())
13                                            {nodes[nodes.length]=elem;}
14  // For each document portion fulfilling the structural condition
15  for(var i=0;nodes.length>i;i++){
16   var isbnNode=nodes[i];
17   // Checking content condition
18   var isbn=isbnNode.innerHTML.match(new RegExp("ISBN-10:</b> (\\d{10})"))[1];
19   if(isbn!=null){
20    // Adding the augmentation lever
21    var link=document.createElement("a");
22    link.innerHTML="Price At BookByte for "+isbn;
23    isbnNode.appendChild(link);
24    (function(_link,_ad,_isbn){
25    // Listen to the user interaction
26    _link.addEventListener("click",function(){
27     //Invoking augmentation service
28     GM_xmlhttpRequest({
29      method: "GET",
30      url: "http://www.bookbyte.com/product.aspx?isbn="+_isbn,
31      onload: function(response) {
32       if(response.status!=200){alert("Error: "+response.statusText);return;}
33       //Rendering service response, encapsulated in "visualize" call
34       var bookBurroPanel=visualize(response.responseText,
35                        "//span[@id='ctl00_ContentPlaceHolder1_lblBestNew']","(.*)");
36       _ad.appendChild(bookBurroPanel);
37      },
38      onerror:function(response){alert("Server Not Exist");}});},true);
39    }(link,isbnNode,isbn));}}},true);
```

Figure 2.6: *BookBurro* in *JavaScript* (partial view).

thousands of uploads (which evidences the effectiveness of the approach). Concretely, 1 script has over 14,000,000 downloads, 10 scripts over 10,000,000 downloads, and 6,000 scripts over 1,000 downloads. Besides downloading, a stronger implication is providing feedback about the own experiences on using scripts. This requires people to take the time to sign up and join the community as members. At the time of this writing, the number of registered users at *userscripts.org* is up to 30,414 users of which 20,000 have commented at least once. Additionally, the number of discussions, replies, or comments are typical ways to measure engagements around specific subjects raised by the own community (e.g. dictionaries, *YouTube*, games). For *userscripts*, some figures follow:

22,157 topics are available which generated 134,690 comments, and over 100 communities of interest. We consider script uploading the ultimate state of engagement. At this regard, *userscripts* enjoys over 34,779 scripts where over 13,500 users have contributed at least once. These figures make us conclude that user scripting is having a profound impact on a large number of users.

2.3.4 Current Research Issues

~~This approach~~ incurs in important drawbacks. **Scripts are vulnerable to page changes**. According to [Fil06], "scripts must often rely on pattern expressions in *XML* query (*XPath*) expressions, but pattern matching can be an easily disrupted technique" and "web page formats evolve", therefore if a web page changes, the script may stop working. Back to our sample case, if the *Amazon* website is upgraded, all "the screen scrapping" can fall apart. For instance, *BookBurro* first retrieves the book's *ISBN* from the current page, and next, injects the *BookBurroPanel* at a certain location. This is normally achieved through *XPath* expressions (see Figure 2.6, line 9). If *Amazon* pages are changed then, *BookBurro's XPath* expressions could no longer recover/identify the right *DOM* node. Therefore, *GM* scripts are specially prone to maintenance. Besides their own maintenance, scripts are affected by the maintenance of the hosting website. The problem is that websites are reckoned to evolve frequently.

Another concern is **script collision**, i.e. the simultaneous access to the same *DOM* node by two different scripts. The very same web page can be subject to different augmentations. *Amazon* is a case in point. At the time of this writing, 268 scripts are reported to be available for *Amazon* at *userscripts*. If you are a regular *Amazon* visitor, it is likely you have several scripts installed. These scripts will be enacted simultaneously when you visit *Amazon*. It is important to notice that script execution is not in parallel but in sequence, i.e. scripts are launched in the order in which they were installed. This implies that the first script acts on the original *DOM* tree,

the second script consults the *DOM* tree but once updated by the first script, and so on. The problem is that programmers develop scripts from the original *DOM*, being unaware of changes conducted by other companion scripts. This can end up in a real nightmare where code developed by different authors with different aims, is mixed up together with unforeseen results. Even worse, the final *DOM* tree can even be dependent on the order in which scripts are enacted! The larger the set of (companion) scripts, the higher the likelihood of clashes. The problem is similar to "scripts are vulnerable to page changes" but now the web page does not change by itself but by the execution of other script. This problem, coupled with the fact that the number of scripts is steadily growing, will likely lead to an increase in the number of scripts in each *user* installation, and hence, in the likelihood of collisions.

The bottom line is that the expressiveness brought by a general-programming language such as *JavaScript* comes at the price of intensive development, and, what is most important, maintenance. Consumption also suffers from this freedom. Even fully-tested scripts (e.g. the *Skype* button) can collide when enacted simultaneously with scripts that access the same *DOM* regions. The problem is that these errors are detected (and suffered) by users with little help from programmers who can hardly foresee the context in which their scripts are to be run. This potentially **high cost of development, maintenance and consumption**, compromises the "end-userness" of *JavaScript* for *Web Augmentation*.

2.4 Conclusions

This chapter gives a brief introduction about the required background to understand the rest of this thesis. *Web Personalization* and *Web Augmentation* are introduced as two different ways to modify web applications and adapt to the user needs. For more detailed information about these areas, see the references of this section.

Chapter 3

Server-Side Open Personalization

3.1 Introduction

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. By its very nature, *Web Personalization* can be one of those parts. 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. 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. Company's website will we enhanced by providing up-to-date user customizations made by its partners with minimal burden. The term *"Server-Side Open Personalization"* is coined to refer to those practises and architectures that permit partners to collaborate in the personalization of the website, namely, permit partners to inject their own personalization rules (known as *"mods"*) in an external company's website.

This chapter is organized as follows. In Section 3.2, this work is motivated using the *ICWE* conference site as an example followed by the research question faced in this chapter. Next, in Section 3.3, the main requirements desired for *Server-Side Open Personalization* architecture are introduced. Then, the existing solutions and our contribution are presented and contrasted with the requirements. In Section 3.4, the *Modding Interface* is introduced as a mechanism to shield *mods* from *"design decisions that are likely to change"* in the website. Based on previously defined requirements, the *Modding Interface* for *Server-Side Open Personalization* is described in Section 3.5, the changes that have to perform the website owner and partners are described in Sections 3.6 and 3.7, and the composition of all pieces are synthetized in the architecture in Section 3.8. Finally, in Section 3.9, the requirements are revised from the proposed solution viewpoint. Conclusions end the chapter.

3.2 Motivating Scenario and Research Question

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 [Bru96]) 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 are 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 user interface (*View*) but also the internals of the application (*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).

Previous scenario serves to illustrate the research question:

> ***How can website owner open its site to their partners allowing the modification of the user interface with low development burden, in a dynamic environment and without compromising the stability and idiosyncrasies of the website?***

The research question is described in depth in the next section, setting the requirements of *Server-Side Open Personalization* architecture.

3.3 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. *Server-Side Open Personalization (SOP)* 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" [O'R04]. 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". *SOP* 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 *SOP*:

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

Next, the previous requirements are put in the context of the existing solutions.

3.3.1 Existing Solutions

The related work in the area is mainly focused on component models and web frameworks. Component model solutions describe the organization and lifecycle of the components of a system. Component models solutions are suitable for systems that require extensibility and resilience. Extensible web frameworks allow the creation of web systems based on modules. Extensible web frameworks are used when extensibility and affordability are the main requirements of a web system.

Component model. The *Open Services Gateway Initiative (OSGI)* [The11] 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. In the same direction, in [Bir05] a pure plugin architecture used by *Eclipse* framework is proposed. Whereas traditional plugin architecture is composed by a hosting application and a set of plugins that extend its functionality, the pure plugin architecture is only composed by a plugin engine and plugins that interacts with ones another. Departing from an already built application, transform to a traditional plugin architecture implies to create a layer that interacts between hosting application and plugins, whereas transforming to a pure plugin architecture implies refactor the already built application. Taking into account the *affordability* requirement, *SOP* has to follow the traditional plugin architecture because it requires less effort create the layer that interacts with plugins than refactor the hosting application. In addition the traditional plugin architecture reflects the asymmetric relationship between the website owner and the third parties and does not allow the interaction/dependency between plugins, which is desirable to a *SOP* because it improves the *resilience*.

Extensible web framework. The ability to respond quickly to rapid changes 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* [JWM10], that resembles *MEF*, the *.NET* library for third-party plugin extensibility. Definitely, *SOP* has to take into account some of the aspects of *PLUX .NET/MEF* to support the extensibility requirement. Notice that *PLUX .NET* does not provide any guide to describe any interface between website core and personalization rules codified in the plugins.

More akin with the *SOP* vision is *SAFE* [RBG12] 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, *SOP* advocate for a more *evolutionary* approach. 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.

Based on the drawbacks of the existing solutions, in the following subsection we describe our contribution in terms of the requirements.

3.3.2 Our Contribution

Server-Side Open Personalization (*SOP*) provides a model of interaction between website owner and its partners, an "architecture of participation". This architecture departs from an existing application, *SOP* only makes the assumption of the use of the *MVC[KP88]* pattern by the website. Our solution instantiates the previous architecture, using a traditional plugin

architecture, code annotation, the *Inversion of Control* pattern and the well-known event-based programming model as the mechanism to support the extensibility requirement. We use existing programming approaches as much as possible, in order to make its use affordable. We explain the instantiation of this architecture taking the *ICWE* website as a running example.

In order to improve the resilience of the architecture, the *Modding Interface* is introduced as a mechanism to shield modifications from *"design decisions that are likely to change"* in a web application. Next section introduces it.

3.4 The Modding Interface

A main principle of Software Engineering is *information hiding* i.e. "the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed" [Par72]. Information hiding then implies the existence of a stable interface which shields consumers from the implementation.

The **Modding Interface** aims at shielding modifications (hereafter **mods**) from *"design decisions that are likely to change"* in a web application. Isolation solutions should be sought to ensure that the evolution of the website has minimal impact on the existing *mods.* From web application viewpoint, design decisions are reflected in the *Model*, *View* or *Controller*. Among these decisions, those related with *Model*, that is formed by domain concepts, tend to be more stable than the other two. Hence, it comes as no surprise that modding should be mainly based on the *Model* of the web application.

Concepts describe meaningful units of what (rather than how) is being rendered by the web application. They attempt to capture the essence of the notions being handled by the website. For instance, *"book"* could be a concept for *Amazon*, *"flight"* for *Expedia*, *"paper"* for *conference site*,

etc. From all of these Concepts, **Modding Concepts** are those Concepts whose rendering realization is amenable to be leveraged by a *mod*.

To ensure decoupling, all interactions between web application and *mods* are conducted through events. This raises the notion of **Publishing Event** and **Processing Event**. A *Publishing Event* (e.g. *LoadBook*) denotes a notification of a *Modding Concept* (e.g. *Book*) being rendered. This event can be consumed by a *mod* through a handler (a.k.a. listener), who can access to the *Modding Concept* through the event payload. A *Processing Event* (e.g. *AddViewModBook*) denotes a request for a modification over a *Modding Concept*. This event can be created and raised by a *mod*.

This event-based mechanism improves the resilience of the *mod* to web application upgrades, but does not preclude the *mod* itself from changing the rendering of the page in "unsafe ways". So far, the *mod* creator can inject any *HTML* fragment on the premise that the disclosure of the page rendering makes him acknowledgeable about what would be the right fragment code. This approach may work in small but is hardly scalable for complex pages. We cannot rely on *mod* creator peering on *HTML* code to ascertain what would be a wrong fragment. This is the role of *Modding Constraint.*

As their database counterparts, a **Modding Constraint** describes a constraint that should be obeyed no matter what and where the modification of the website is achieved. These constraints reflect invariants on the layout or aesthetics concerning a given *Modding Concept*. For instance, modifications of the concept *"Book"* should be compliant with the following constraints: if the layout is extended then, the supplemented code is restricted to be of type *HTMLParagraphElement* as defined by the *W3C*. Now, a *mod* that raises a *"AddViewModBook"* must generate an *HTMLParagraphElement-compliant* fragment.

As a proof of concept, next section introduces *Modding Interface* for *Server-Side Open Personalization*.

3.5 Specification of the Modding Interface

SOP 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*. The isolation solution proposed is based on the usage of the previously introduced *Modding Interface.* As previously stated and applying the terms to *.NET* framework, **Modding Interface** can be based on *Model Classes (Model)*, *Web Forms (View)* or *Controller classes (Controller)*, but Model classes are chosen because they 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*. As previously stated, 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 [JCP03].

- **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 *SOP* for owners and partners alike. This is the topic of the next sections.

3.6 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 3.1 shows the case for the ICWE website: the class *Accommodation* becomes a Modding Concept. *[ModdingConcept]* 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 [W3C00b]. For instance, modding an *"Accommodation"* is set to be of type *HTMLTableCellElement,* meaning that mods to *Accommodation* need

```
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  }
```

Figure 3.1: Domain classes annotated to become *Modding Concepts*.

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

```
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>
```

Figure 3.2: Mod-aware Views: the ASPX includes a place holder that accesses the *AccommodationMod* (line 8).

for this situation, though this is up to the host. Figure 3.2 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 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*.

3.7 Impact on Partners: Defining Mods

Unlike the open-source approach, *SOP* 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*.

```
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    }}
```

Figure 3.3: *Mods* as plugins that import *Modding Interfaces* (line 8).

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 3.3 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":*

- 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

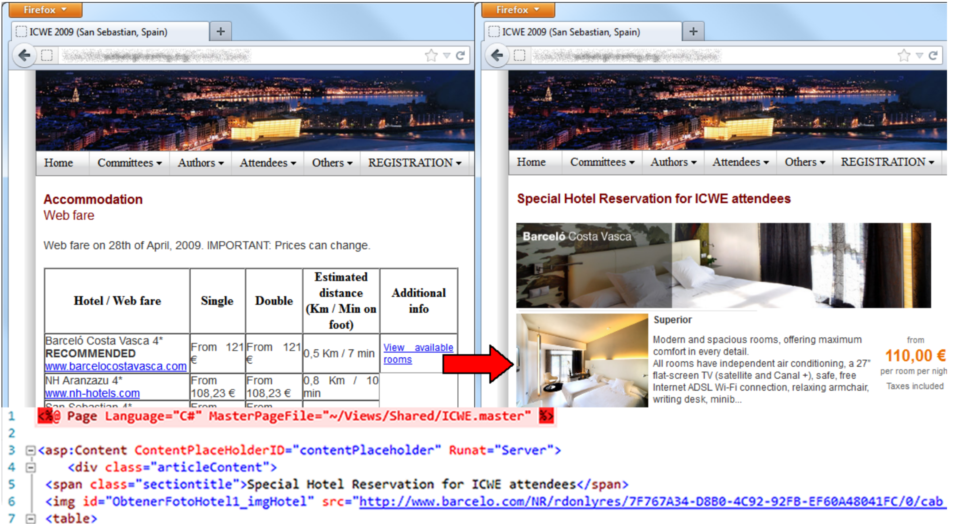

Figure 3.4: A *mod* that introduces a new *View* & *Controller*. In the up side, the host's View links to the partner's *View* and the rendering of the partner's *View*. In the down side the partner's *View* code refers to the host template (i.e. MasterPageFile).

are obtained dynamically using dependency injection (see next subsection). 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*.

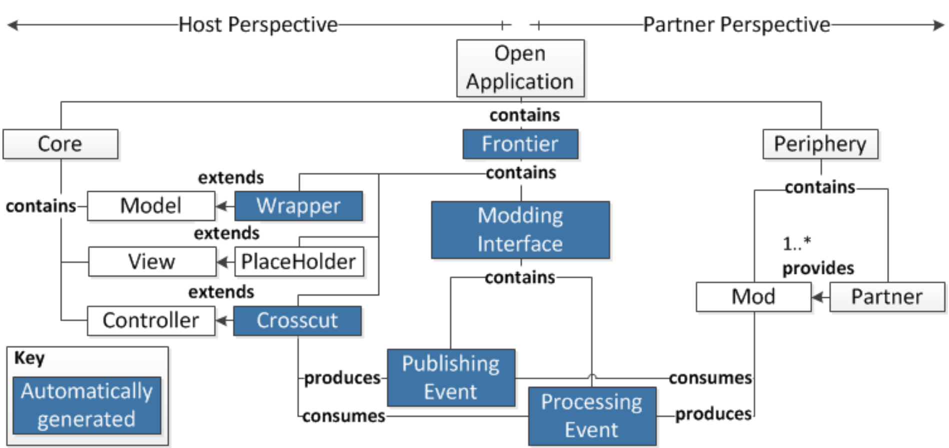

Figure 3.5: Decoupling the *Core* from the *Periphery*: a model of the involved concepts.

Adding New Views & Controllers. In the previous example, the output of the *mod* could have contained links to *Views* with additional information (e.g. room pictures). Figure 3.4 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 3.4). This permits the partner's *Views* to link back to the rest of the website.

3.8 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 3.5 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 [Fow04]. *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.

3.9 Discussion

In this section, the four requirements, namely, resilience, extensibility, scalability and affordability are revisited and evaluated in terms of the proposed solution.

3.9.1 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.

3.9.2 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.

3.9.3 Scalability

Mods should not deteriorate the site performance. *SOP* 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 subsection describes the results of a stress testing (a.k.a. load testing) of the *SOP* 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

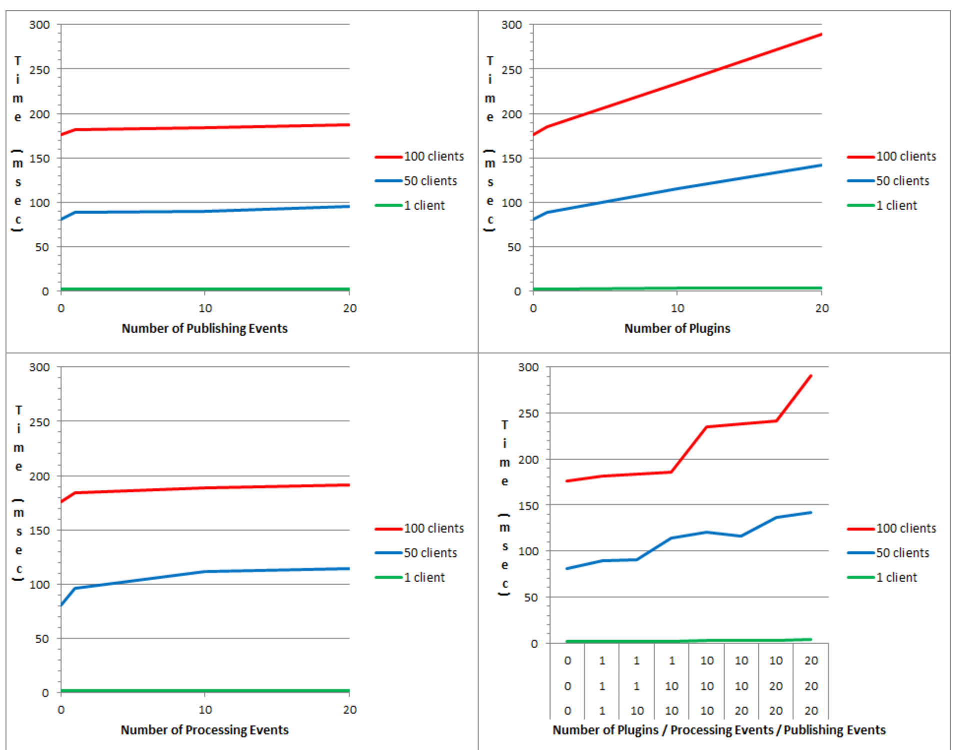

Figure 3.6: Latency introduced by distinct *SOP* factors (clockwise from bottom left): #Processing Events, #Publishing Events, #Plugins, and finally, the combined effect of all three.

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 [FR03]. *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.

[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.

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 3.6 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, *SOP* 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 *SOP* architecture represents around a 2% of increment for the most common envisaged scenarios.

3.9.4 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. *Publishing Events*). This certainly leads to cleaner code. On the downside, the flow of the program is usually less obvious.

3.10 Conclusions

Fostering a win-win relationship between website owners and partners, substantiates the efforts of *Server-Side Open Personalization (SOP)*. *Server-Side Open Personalization (SOP)* 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 *SOP* 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, *SOP* 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 chapter however, we focus on the technical feasibility of *SOP*. The proposed solution is resilient to changes on the website, website evolution does not cause any crash but the no rendering of the *mod* in the worst case. Extensibility is achieved using the *Modding Interface* and an event-based mechanism, and realized using the *Inversion of Control* pattern. Scalability is tested, in order to ensure that this architecture does not suffer of a noticeable deterioration in performance; therefore there are no needed additional resources to host the website. The architecture is based on *.NET MEF*, code annotations, the *.NET* event mechanism and software interfaces as non disruptive programming practices to make the development of the solution affordable for website owner and partners. Website owner mainly need to annotate the Model classes/attributes to automatically generate the *Modding Interface*. The partners only need the *Modding Interface* to know which are the parts disclosed of the Model namely the *Modding Concepts*, how to access them subscribing to *Publishing Events* and how to communicate the modifications notifying

Processing Events. Though proving feasibility requires focusing on a specific platform, the approach is easily generalizable to any framework that supports *Inversion of Control*.

Chapter 4

Hybrid Open Personalization

4.1 Introduction

Web Augmentation alters the rendering of *existing* web applications at the back of these applications. Changing the layout, adding/removing content or providing additional hyperlinks/*widgets* are examples of *Web Augmentation* that account for a more personalized user experience. This is achieved through *JavaScript (JS)* using browser weavers (e.g. *Greasemonkey*). To date, over 43 million of downloads of *Greasemonkey* scripts ground the vitality of this movement. Such augmentations are created and used only by the *Greasemonkey* community. The term *"Hybrid Open Personalization"* is coined to refer to end users not only the beneficiaries but also the contributors of augmentation scripts. Unfortunately, current development models offer little help in understanding and managing this new form of value co-creation. The *Metropolis Model* [KC09] has recently identified three realms of roles for *commons-based peer production* (crowdsourcing [How06]): the kernel

(providing the core functionality), the periphery (the scripters) and the masses (the end users). The periphery requires mechanisms for the commons to suggest, develop and maintain additional services on top of the kernel. This work concretizes the *Metropolis Model* for crowdsourced website development based on user scripts. We outline some challenges to foster the relationship between end users (the masses), scripters (the periphery) and the web site (the kernel) on the way to promote script-based crowdsourcing.

This chapter is organized as follows. In section 4.2, *Hybrid Open Personalization* is motivated using the *ICWE* conference site augmented with information from *DBLP*. Next, in Section 4.3, the main requirements desired for *Hybrid Open Personalization* architecture are introduced. Then, the existing solutions and our contribution are presented and contrasted with the previous requirements. In Section 4.4, *Crowdsourcing Web Augmentation* is illustrated using the *Metropolis Model* to introduce the existing roles. Once these roles are put in the web setting, the scripting main issues are presented: development, testing, advertising and sandboxing. Section 4.5 and 4.6, introduces the *Modding Interface* mechanism, that shelters scripts from changes in websites but now adapted to client-side scripting. In section 4.7, the mod development using *Modding Interface* is introduced and compared with traditional development. Once the script is developed, it can be tested using *Modding Contracts* as described in section 4.8. Section 4.9, describes script advertisement and section 4.10 propose and architecture to safely execute the scripts. Finally, in Section 4.11, the requirements are revised from the proposed solution viewpoint. Conclusions end the chapter.

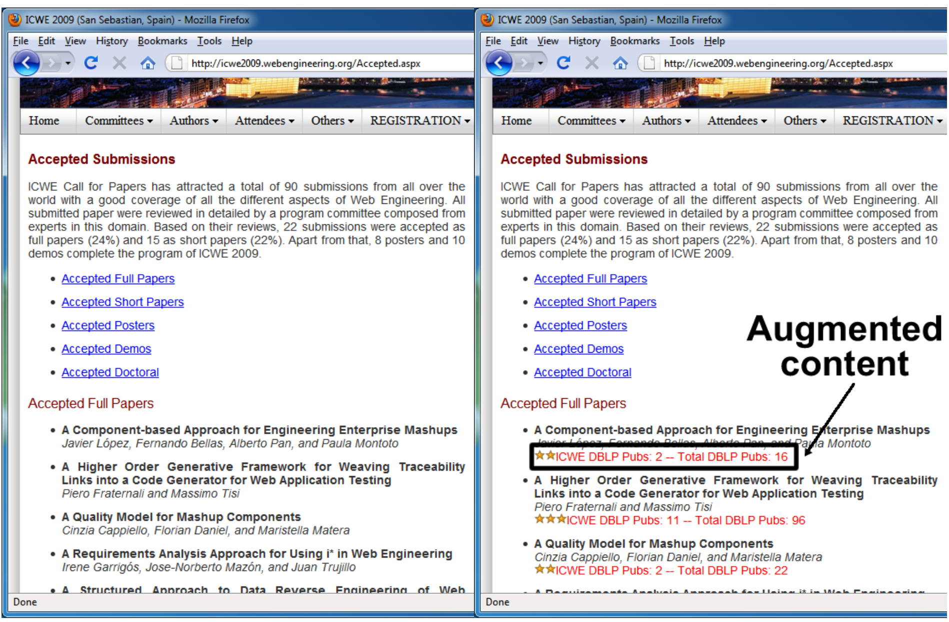

Figure 4.1: Raw page (left side) vs. Augmented page (right side).

4.2 Motivating Scenario and Research Question

Conferences addressing web issues can tap on their attendees to enhance the conference site itself. The vision is to regard the conference site as a platform for attendees to enhance. As an example, consider the conference website for *ICWE'09*. Figure 4.1 (left side) depicts a screenshot for the page on accepted submissions, located at *http://icwe2009.webengineering.org/Accepted.aspx*. On deciding which presentations to attend, an attendee can augment this content with data obtained from *Michael Ley's DBLP* site[1] so that each accepted paper is augmented with data about previous publications from the paper's authors. To this end, the attendee writes the *dblpFigures* script[2] (see Figure 4.2, left side). The outcome (see Figure 4.1, right side) shows how *"host markup"*

[1] http://dblp.uni-trier.de/db/index.html

[2] Available at *http://userscripts.org/scripts/source/76472.user.js*.

```
1  // ==UserScript==
2  // @name          Simplified version of dblpFigu
3  // @description   DBLP past publications for pap
4  // @include       http://icwe2009.webengineering
5  // ==/UserScript==
6
7  var doc=window.document;
8
9  //MOD-LOGIC
10 function createDblpFiguresPanel(author){
11  var dblpFiguresPanel=doc.createElement("span");
12  //A panel with a past publications of selected
13  //author from DBLP is created and assigned to
14  //the dblpFigures variable
15  ...
16  return dblpFiguresPanel;
17 }
18
19 function init(){
20  //HTML SCRAPING
21  var papers=document.evaluate(
22 "//*[@class='paper']",document,null,
23 XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,null);
24  for(var i=0;i<papers.snapshotLength;i++){
25   //HTML SCRAPING
26   var firstAuthor=document.evaluate(
27 ".//*[@class='author']",papers[i],null,
28 XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE,null).
29 item(0);
30  //MOD-LOGIC CALL
31  var dblpFiguresPanel=
32  createDblpFiguresPanel(firstAuthor);
33  //HTML INJECTION
34  papers[i].appendChild(dblpFiguresPanel);
35  }
36 }
37                                         DOM Event
38 doc.addEventListener('load',init,true);
```

```
1  // ==UserScript==
2  // @name          Mod version of dblpFigures
3  // @description   DBLP past publications for pap
4  // @include       http://icwe2009.webengineering
5  // ==/UserScript==
6
7  var doc=window.document;
8
9  //MOD-LOGIC
10 function createDblpFiguresPanel(author){
11  var dblpFiguresPanel=doc.createElement("span");
12  //A panel with a past publications of selected
13  //author from DBLP is created and assigned to
14  //the dblpFigures variable
15  ...
16  return dblpFiguresPanel;
17 }
18
19 function init(loadPaperOcc){
20  //EVENT PARAMETER RETRIEVAL
21  var paper=loadPaperOcc.currentTarget;
22  var firstAuthor=
23  paper.getElementsByTagName("author").item(0);
24  //MOD-LOGIC CALL
25  var dblpFiguresPanel=
26  createDblpFiguresPanel(firstAuthor);
27  //EVENT DISPATCH FOR HTML INJECTION
28  var appendChildPaperOcc=
29  doc.createEvent("ProcessingEvents");
30  appendChildPaperOcc.initProcessingEvent(
31     "appendChildPaper",paper,dblpFiguresPanel);
32  doc.dispatchEvent(appendChildPaperOcc);
33 }
34
35
36                                            Conceptual Event
37
38 doc.addEventListener('loadPaper',init,true);
```

Figure 4.2: Two versions of the *dblpFigures* script: using *DOM Events* (left side) vs. using *Conceptual Events* (rigth side).

is intermingled with *"augmented markup"* produced by the script.

This process takes place at the client. The hosting application is completely unaware of this process: no responsibility is taken on certifying or disseminating augmentation scripts among its users. Script safety is not validated, hence, script users are exposed to malware.

This is certainly bad news for users but so is it for web application owners. Although augmentation can threat the business models of some sites (e.g. by removing banners), in other cases, augmentation accounts for honest enhancements that serve a small set of users the application cannot afford to support their requirements, but leaves external users to fill the gap. After all, popular sites such as *Facebook*, encourage their users to build and share *Facebook applications* on the certitude that this increases

the stickiness and usefulness of the site [Fac10a, MP09]. Customer loyalty, engagement and satisfaction are among the rewards. The vision is to create an open ecosystem between the hosting application and the augmentation contributors. This leads us to the research question:

> ***How can website support, promote and coexist with user-provided scripts?***

The research question is described in depth in the next section, setting the requirements of *Hybrid Open Personalization* architecture.

4.3 Requirements

This chapter outlines some technical challenges supporting, promoting and enabling the coexistence of augmentation contributions, namely: (1) insufficient decoupling between user scripts and the underlying website (i.e. the platform); (2) no means for website customers to know about user scripts; (3) lack of mechanisms for websites to certify scripts; and (4), lack of an architecture to ensure the integrity of the website.

Based on these observations, we introduce the following quality criteria (and driven requirements) for "an architecture of participation" for *Hybrid Open Personalization (HOP)*:

- **Affordability**. Contributors effort should be minimized. All the steps of the *mod*'s lifecycle, from development to advertising, have to be considered. Designs based on widely adopted programming paradigms stand the best chance of success. The more contributors you expect to attract, the simpler it must be and the more universal the required tools should be. The hosting web also needs to ensure the soundness of the contributors' *mods*.

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

- **Scalability**. Growing amount of *mods* should be handled in a capable manner.
- **Security**. *Mods* should not be able to do malicious tasks like redirection to phishing pages or stealing user sensitive data.

4.3.1 Existing Solutions

This work is in the middle of two existing approaches, application centric and user programming. Application centric approaches allow the extension of a concrete application. This kind of solution takes into account the peculiarities of its application and normally cannot be reused in other contexts. Application-centric approaches mainly take care of the resilience and security concerns. User programming approaches enable end users to customize the web without the involvement of the webmaster. This approach prioritizes affordability as the key requirement.

Application centric. Distinct websites use a back-end approach (i.e. *API*-based) to open up their platforms. *Amazon* is among the best known ([IL10]). However, fewer experiences exist on using a front-end approach. A hybrid example is provided by the *Facebook Developer Platform*, launched in 2007 ([Fac10b]). This platform provides three enablers: (1) a *REST API* to access data profile, friends, photos, etc; (2) the *Facebook Query Language (FQL)*, which mimics *SQL*-like syntax to achieve a similar functionality to that of the *API*, and (3), the *Facebook Markup Language (FBML)*, a markup language that can be interlinked with your own *HTML*. These mechanisms permit end users to create their own applications outside *Facebook*. Once developed, the application is deployed at the user side but needs first to be registered in *Facebook*. Notice that neither *Amazon* nor *Facebook* allow web augmentation; only *Facebook* offers a front-end approach but in a side-by-side way.

A related initiative is that of *Facebook*'s Social Plugins ([Fac10c]). Plugins are served by *Facebook* itself as a means to personalize your *Facebook* account. Contributors are websites that inlay these plugins.

Normally, this is rendered as an icon on the website's page. On clicking the icon some data flows to your *Facebook* account. In this way, websites become data contributors for *Facebook*. The *"Like Button"* is among the most popular plugins. When end users click the *Like* button, a link to the contributor page is added to the user *Facebook* profile, and a story is shared with the user's friends.

User programming. Mashup tools (e.g. *Yahoo! Pipes* ([Yah07]) are classified as end-user programming, web-based ecosystems ([Bos09]). Mashups and *mods* have a lot in common. They both focus mainly "on opportunistic integration occurring on the web for an end user's personal use and for non business-critical applications" ([YBCD08]). However, modding and mashups differ in the aim. Mashups are akin to *integration* efforts to build *new applications* out of existing resources. This is, most of the examples so far aggregate data coming from different sources which end up conforming an application in its own right, detached from the source websites (*Yahoo Pipes* is a case in point). By contrast, modding does not produce a new application but enhances an existing website. A *mod* can only be understood with reference to the application it tunes. Mashups look at websites as feeding sources while modding regards websites as platforms.

Operator is a plug-in for *Firefox* that detects *microformats* in the current page which are then displayed through a toolbar ([Kap06]). Some *"actions"* can then be associated with the detected annotations, e.g. if an *hCalendar* annotation is spotted then, an action can be introduced to add this event to your calendar; if an *hAddress* annotation then, you can attach an action to place this address in *Google Maps,* and so on. Therefore, *Operator* can be regarded as a general mechanism for seamless data passing based on *microformats* which also leads to an augmented experience. *Operator* provides an example of the advantages brought by having well-structured websites. After all, the rationale behind *microformats* is improving page structure in a well-focused but limited area (e.g. calendar information). However, the number of things that can be

described by *microformats* is limited (i.e. they are not infinitely extensible and open-ended).

4.3.2 Our contribution

Hybrid Open Personalization (*HOP*) provides a model of interaction between website owner and augmentation contributors. Our solution is composed by a set of models: *Modding Interface* that improves the resilience between website and *mods*, *Modding Contracts* as a mechanism to certify the soudness of the *mods*, and the sandboxing of the *mods* to ensure the security of the system. In order to promote the dissemination of the *mods*, the preview mode is introduced. We explain the instantiation of those models taking the *ICWE* website as a running example.

The rest of this chapter introduce a *HOP* using *Web Augmentation* techniques (a brief introduction to *Web Augmentation* is in Section 2.3).

4.4 Crowdsourcing Web Augmentation

The term "*crowdsourcing*" has been coined to denote the shift to commons-based peer production, viewing customers not as passive users but as co-creators of value. Unfortunately, current development models offer little help in understanding and managing this new form of value co-creation. The *Metropolis Model* has recently identified some issues for crowdsourcing. This work concretizes the *Metropolis Model* for crowdsourced website development. In Figure 4.3, this *Metropolis Model* is represented. This model distinguishes three realms of roles: kernel, periphery and masses. The kernel provides the core functionality and content of the website, and traditional methodologies can be used. However, the periphery requires mechanisms for the commons to suggest, develop and maintain additional services on top on the kernel, the *mods*.

An open ecosystem implies for the platform to take a more active role in supporting:

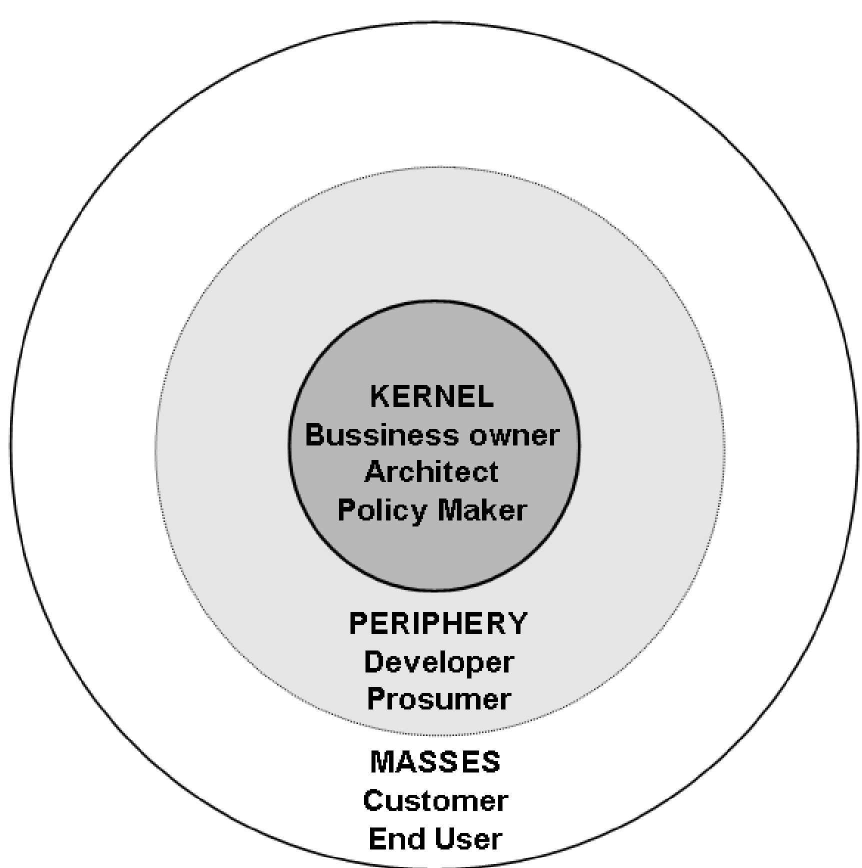

Figure 4.3: The Metropolis Model. The periphery extends the application's ecosystem.

- script development, interaction between kernel and periphery. This implies proper architectural decoupling between the website and the user scripts. So far, scripts are fragile to website upgrades. Changes in the rendering of the website can make scripts fall apart. And websites are reckoned to change frequently. This work introduces the notion of *Modding Interface* as a means to shelter scripts from changes in the underlying website,

- script testing, interaction between kernel and periphery. Ecosystem development should not imply a decrease in the quality of the final product. Mechanisms need to be introduced that allows community-

offered scripts to exist within the platform but with no impact on the core functionality to any significant extent. That is, websites should certify the soundness of user scripts. To this end, we introduce a test battery for scripts. This battery is provided *by the website* for user scripts to satisfy before being publicized among website customers,

- script advertising, interaction between periphery and masses. So far, script finding is achieved through script repositories. These repositories act as yellow pages which offer general information about scripts. In most cases, the user is forced to install the script to see what the script looks like. This difficulty in both finding and understanding the script are detrimental for the end user but also for both the script programmer and the website. To this end, we introduce a second mode for websites to operate: *the preview mode*. In the traditional mode, users get the core functionality, i.e. the website recovers the base pages. By contrast, the preview mode permits to visualize base pages but now augmented with selected scripts included in the platform offerings. The preview mode does not require any additional installation to facilitate user engagement,

- script sandboxing, interaction between periphery and masses. *Crowdsourced* augmentation implies *mod* code (i.e. the scripts) to co-exist with hosting code (e.g. the *HTML page*). This is risky. Placing different resources for numerous and possibly untrusted or malicious sources into the same security domain raises security and integrity concerns. Threats include: creation of/redirection to phishing pages, stealing history information (or sensitive data stored on either pages or cookies), or port scanning upon the user's local network [Goo]. It is needed a mechanism to protect the user from malicious *mods*. To this end, we offer an architecture for safe co-existence of hosting code and augmentation scripts. This architecture is based on *Modding Interface*. Set and managed by the hosting application, this interface regulates both the outflow

(i.e. what "hosting data" can flow to augmentation scripts) and the inflow (i.e. what "hosting rendering aspects" can be subject to augmentation). Traditionally, scripts have open access to hosting rendering through *DOM events*. Now, scripts are "sandboxed" so that interaction can only be through *MI events*.

In the next section, the notion of *Modding Interface* is revisited and adapted to the scripting world.

4.5 The Modding Interface: a Client-Side Perspective

Crowdsourcing implies encouraging contributor participation. In our setting, this implies sheltering scripts from upgrades in the underlying website. To this end, we propose the notion of *Modding Interface* that was previously introduced in the previous chapter, in the section 3.4 in the context of *SOP*. Next paragraphs explain this mechanism that was adapted to the *HOP* context.

Interfaces are commonly specified in terms of operations defined upon data types. However, *JavaScript* favours event-based programming, i.e. handlers are associated to UI events. Unlike operations, handlers are not explicitly called but *triggered* when the associated event occurs. Akin to the *JavaScript* approach, *Modding Interfaces* are to be described in terms of events rather than operations, but they will act upon *concepts* (e.g. *Paper*) rather than *DOM nodes*. In this way, scripts can subscribe to the event *loadPaper* (rather than the *DOM event*, *load*) and obtain *Paper* data as event payload rather than scraping the *DOM tree*. Scripts can also publish the event *appendChildPaper* to add an *HTML fragment* as a child of a *Paper* (rather than using an *XPath expression*).

The right side of Figure 4.2 shows the *dblpFigures* augmentation script but now using *Conceptual Events*. The augmentation logic is the same (lines 10-17). Differences rest on (1) *HTML* scraping being substituted

by event parameter recovering (lines 21-23) and, (2) amendment spaces described by the point where *Conceptual Events* occur (lines 28-32) rather than *XPath expressions*.

Therefore, a *Modding Interface* encapsulates a web application in terms of its concepts, and provides a set of services to "read" and to "write" these concepts. The "read" part realizes the required interface as the set of events the interface just signals but leaves to the scripts the event processing (a.k.a. **Publishing Events**). As for the "write" part, it identifies the amendment space in terms of concept occurrences rather than through *DOM nodes*. Rather than using *XPath* on *DOM trees*, the amendment space is identified by the target of **Processing Events**. For instance, the event *appendChildPaper* indicates that *Paper* denotes an amendment point. Scripts can now raise *appendChildPaper* to inject its *HTML markup* into this amendment point. *Processing Events* then realize the provided interface[3].

Next section describes how the concepts of the *Modding Interface* model are specified.

4.6 Specification of the Modding Interface

The *Modding Interface* is described as an *OWL* document [SWM04]. *OWL* permits to describe concepts, properties related to these concepts and associations between concepts. The aim of *OWL* is to provide a way to exchange information between applications with a specific semantic. Such aim aligns with our purposes.

Modding Interfaces are described through *"Concepts", "PublishingEvents"* and *"ProcessingEvents"* instances[4]. Next paragraphs describe each notion, using a conference website as an example (see

[3]The terminology of "processing events" and "publishing events" is widely used for event-based components such as portlets [JCP03].

[4]A schema for defining *Modding Interfaces* is available at *http://userscripts.org/scripts/source/61129.user.js*.

```
1  <!DOCTYPE rdf:RDF [
2  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
3  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
4  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
5  <!ENTITY owl "http://www.w3.org/2002/07/owl#">
6  <!ENTITY swc "http://data.semanticweb.org/ns/swc/ontology#">
7  <!ENTITY swrc "http://swrc.ontoware.org/ontology#">
8  <!ENTITY mod "http://userscripts.org/scripts/source/61129.user.js#">
9  <!ENTITY icwe "http://icwe2009.webengineering.org/">
10 ]>
11 <rdf:RDF xmlns:base="&icwe;" xmlns:owl="&owl;" xmlns:rdf="&rdf;"
12 xmlns:rdfs="&rdfs;" xmlns:mod="&mod;">
13 <owl:Ontology>
14  <owl:Class rdf:about="&swc;ConferenceEvent"/>
15  <owl:Class rdf:about="&swc;Paper"/>
16  <owl:Class rdf:about="&swrc;Person"/>
17  <owl:DatatypeProperty rdf:about="&swrc;title">
18   <rdfs:domain rdf:resource="&swc;Paper"/>
19   <rdfs:range rdf:resource="&xsd;string"/>
20  </owl:DatatypeProperty>
21  <owl:ObjectProperty rdf:about="&swrc;author"/>
22   <rdfs:domain rdf:resource="&swc;Paper"/>
23   <rdfs:range rdf:resource="&swrc;Person"/>
24  </owl:ObjectProperty>
25 </owl:Ontology>
26
27  <mod:PublishingEvent rdf:ID="loadPaper">
28   <mod:payloadType rdf:resource="&swc;Paper"/>
29   <mod:uiEventType rdf:resource="&mod;load"/>
30   <mod:cancelable>false</mod:cancelable>
31  </mod:PublishingEvent>
32
33  <mod:ProcessingEvent rdf:ID="appendChildPaper">
34   <mod:payloadType rdf:resource="&mod;HTMLSpanElement"/>
35   <mod:operationType rdf:resource="&mod;appendChild"/>
36   <mod:targetConcept rdf:resource="&swc;Paper"/>
37  </mod:ProcessingEvent>
38 </rdf:RDF>
```

Figure 4.4: *ICWE Website Modding Interface*.

Figure 4.4).

Concepts.

The *ICWE* Website is seen as renderer of a set of concepts: *ConferenceEvent*, *Paper*, *Person*, etc. (lines 14-16). The <*Ontology*>

element contains the description for these concepts. Concept description includes <*DataTypeProperty*> and <*ObjectProperty*> (i.e. title and author (lines 17-24)). It is possible to import the ontology. For conference description, an external ontology is available at [MBH07].

Publishing Events.

These events notify "concepts" being delivered by the web application. In other words, the payload of a *Publishing Event* is a concept of the web application at hand. But events are happenings of interest, i.e. they are instants of time. An event cannot be described just by its associated concept but needs to include what happens to this concept, e.g. loading, selecting, de-selecting the concept, etc. Therefore, *Publishing Events* are described by both the event payload (*"**payloadType**"* property), and the time when the event arises (*"**uiEventType**"* property). The values for *"uiEventType*" are taken from the *W3C's DOM Level 2 Events* specification [W3C00a]. Additionally, a *"**cancelable**"* property is added that mimics the namesake property available for *JavaScript* events whereby an event is liable to be called off by a handler so that the occurrence is no longer propagated to other handlers. As specified in Figure 4.4 (lines 27-31), *loadPaper* is introduced as a *Publishing Event* to occur every time a *Paper* is loaded.

Processing Events.

A website determines *what* can be augmented but leaves to the scripter to decide *when* and *how* is to be augmented. The *what* refers to the concept that denotes the amendment space ("***targetConcept***" property). But being a layout issue, the concept alone is not enough. We need to indicate the position w.r.t. the concept ("***operationType***" property) through a reference to the *W3C's DOM Level 2 Core* operations [W3C00a]. Figure 4.4 shows an example where the concept *Paper* is used to pinpoint the amendment space. The *"operationType"* indicates that augmented content is to be

rendered as children of the *Paper* at hand (i.e. *appendChildPaper* (lines 33-37)).

As for the *how*, traditional scripts can inject any *HTML fragment* on the premise that the disclosure of the page implementation makes them acknowledgeable about what would be the right fragment code. This approach may work for simple pages but is hardly scalable as pages become more complex. We cannot rely on end users peering on *HTML code* to ascertain what would be a wrong fragment to be injected. We resort to *HTML types* [W3C00a]. The augmentation markup should be compliant to an *HTML type* ("***payloadType***" property). This type restricts how rendering can be augmented. For instance, augmenting a *"Paper"* is set to be of type *HTMLSpanElement,* meaning that augmentation markup on *Papers* need to be compliant with this type. This introduces a type-like mechanism for regulating augmentation to existing Web application. The *weaver* can then check whether this *payloadType* is fulfilled, and if not so, ignores the script markup but still renders the rest of the page. This is akin to browser practices where wrong *HTML tags* do not prevent the browser from rendering the page.

Once the *Modding Interface* is specified, the scripters can develop *mods* on top of it. The next section goes about this issue.

4.7 Script Development

This section addresses the definition of mod scripts based on *Modding Interfaces*. Our contention is that *Modding Interfaces* causes minimum disturbance on script programming. To this end, native *JavaScript* mechanism is used to notify/publish conceptual events with no variations w.r.t. traditional script development.

Notification of Processing Events. *JavaScript* follows an event-based approach where listeners can be associated with *DOM*-based events. An event is a happening of interest. Event types include: *MouseEventTypes* (e.g. *click, mouseover, mousemove...*), *UIEventTypes* (e.g. *DOMFocusIn,*

DOMFocusOut and *DOMActivate*), *MutationEventTypes* (e.g. *DOMSubtreeModified, DOMNodeInserted*) and *HTMLEventTypes* (e.g. *load, change*). Operations are available for creation of event occurrences (e.g. *createEvent("MouseEvents")*), assigning the payload to an occurrence (e.g. *initMouseEvent("eventInstance", "eventParameters")*), or raising the event manually (e.g. *dispatchEvent(eventOccurrence)*). The following code simulates a click on a checkbox:

```
var ev=document.createEvent("MouseEvents");
var cb=document.getElementById("checkbox");
ev.initMouseEvent("click", true, true, window, 0,
    0, 0, 0, 0, false, false, false, false, 0, null)
    ;
cb.dispatchEvent(ev);
```

The snippet illustrates the pattern for dispatching an event occurrence: [*createEvent*, obtain DOM node, *initMouseEvent*, *dispatchEvent* on this node]. This is standard *JavaScript* code.

Conceptual events mimic this pattern. Back to our running example, a *dblpFiguresPanel* (i.e. an *HTML* fragment) is to be injected as a child of a *Paper*. For this case, the pattern goes as follows: [*createEvent*, obtain concept, *initProcessingEvents*, *dispatchEvent* on this concept]. The code follows (the complete mod script can be found at Figure 4.2 (right side)):

```
var ev=document.createEvent("ProcessingEvents");
var paper = loadPaperOcc.currentTarget;
ev.initProcessingEvent("appendChildPaper",
paper, dblpFiguresPanel);
doc.dispatchEvent(ev);
```

The only difference with traditional scripting is that now injection points are not DOM nodes but the current concept. This current concept is to be obtained through *Publishing Events*.

Subscription to Publishing Events. *JavaScript* achieves event subscription by registering a listener through the *addEventListener* method. An example follows:

```
function init(...){...}
var cb=document.getElementById("checkbox");
cb.addEventListener("click",init ,true);
```

This code associates the script function *init()* with the occurrence of clicks on a *checkbox* node (a.k.a. the event target). From then on, a click on a checkbox will cause *init()* to be enacted. Since most *JavaScript* events are UI events, event occurrences are generated while the user interacts with the interface, raised by the *JavaScript* engine, and captured and processed through script functions.

Subscription to conceptual events is accomplished in the very same way: associating a listener. For instance, instruction (line 38 in Figure 4.2 (right side)) *"doc.addEventListener("loadPaper",init,true)"* adds a listener to the *loadPaper* event, i.e. occurrences of *loadPaper* will trigger the *init()* function. The difference rests on listeners being associated to the whole document (i.e. variable *doc*) rather than to *DOM* nodes (e.g. a *checkbox*). This highlights the fact of events happening on *Papers* rather than on *DOM* nodes that are the circumstantial representation of these *Papers*.

4.8 Script Testing

Website customers are the final beneficiaries. The base experience can now be augmented through mod scripts designed by users for users. Customer loyalty, engagement and satisfaction are among the benefits for the website. However, these benefits are not without risk.

Good mod scripts boost satisfaction. But bad mod scripts can have the opposite effect. So, mechanisms are needed to ensure sound mod scripts before being released to the general public.

Traditionally, 3rd parties are to be certified before granting access to the platform. However, it has been reported that this approach does not work on open ecosystems. As stated in ([BBS10]) "the traditional certification approach is infeasible in this context, especially as the typical

case will contain no financial incentive for the community contributor and the hurdles for offering contributions should be as low as possible. Consequently, in these cases, a mechanism needs to be put in place that allows software to exist within the platform but to be sandboxed to an extent that minimizes or removes the risk of the community-offered software affecting the core problem to any significant extent." This problem goes beyond mod scripts to embrace other Web2.0 platforms. For instance, *Facebook* encourages end users to develop web plug-ins, and shares these plug-ins across the whole site. The problem is that users are free to release their mod scripts at any time without required to pass formal testing ([YLZ09]). As a result, failures and missfunctions occur at post-release time, damaging users and *Facebook* alike. To ensure the quality of plug-in, both industry and research communities have presented various methods. For instance, *Facebook* outlines a list of prohibited plug-in categories and seek for law protection ([Fac10a]).

Again open ecosystems change how testing is conducted. Traditionally, the developer is responsible for the design, implementation and execution of the test cases. In the ecosystem approach, the platform (i.e. the website) is at least as interested as the component developer to ensure the safety of the component. This first implies the existence of a clear interface between the platform and the community components. And second, an active involvement of the platform in defining the test cases to be passed for the component to be certified (i.e. for the mod to be available through the website). Additionally, this process should minimize the hurdles for offering contributions.

To this end, we introduce the *Modding Contract.* This contract states script functionality in terms of the causal relationship between event subscriptions and publications. These contracts are located at the script's metadata block. For instance, the contract: *[loadPaper -> appendChildPaper]* states that each *loadPaper* event will cause an *appendChildPaper* to be signalled. It can be understood as follows: if the pre-condition *"loadPaper is raised"* is satisfied then**,** the postcondition

"the script will raise an appendChildPaper event" will be ensured.

Unit testing can be used to check that a script meets its contract assuming its subcontractors meet theirs (i.e. the website generated appropriate *Publishing Events*). However, unit-testing design is time consuming and requires knowledge about how to obtain full coverage of test cases. This puts an extra burden on the script developer. On the other side, it is in the own interest of the website to thoroughly validate the user scripts. On these grounds, our approach leaves to the platform (i.e. the website) the duty of defining the test units. Scripts need first to be verified against these test units. Only if test passes, the user script is publicized, and liable to be installed.

Design by Contract is an approach to designing computer software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components ([Mey97]). Mod scripts are now the software components. Applying Design-by-Contract to mod scripts then implies: (1) a contract language for mod scripts, (2) an environment for interpreting and validating the contracts, and (3) a mechanism for generating test cases. Next subsections delve into the details.

Contract specification. Having dynamic language features (dynamic typing), *JavaScript* makes it harder to specify the intended behaviour of a system and to demonstrate that the system adheres to a specification. This explains the very few works that address contract specification for *JavaScript* programs ([HT10]). Whereas this is true in general, our proposal for mod scripts rests on the existence of a *Modding Interface*. That is, mod scripts subscribe and publish a restricted set of events: those specified at the *Modding Interface*. We propose to make this explicit through a contract.

A *Modding Contract* states the mod script functionality in terms of the causal relationship between subscriptions and publications. These contracts are located at the mod script's metadata block. As an example, consider the *dblpFigures* mod script (see right side of Figure 4.2 line 5). It

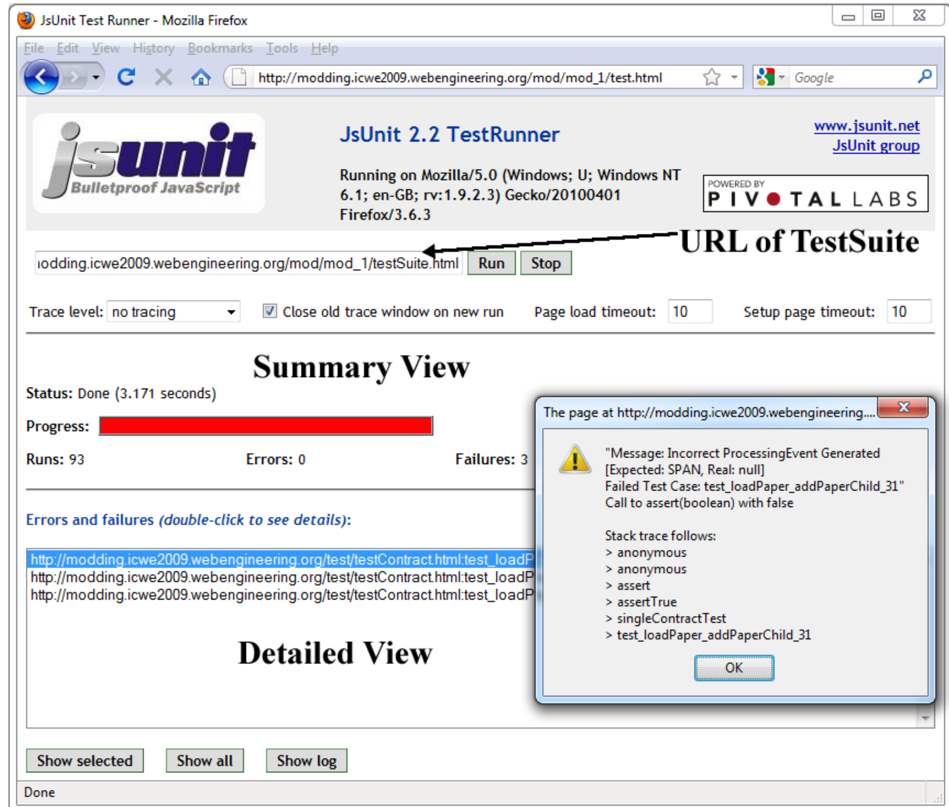

Figure 4.5: Testing *dblpFigures* mod script through the *JsUnit* framework. Three failures are detected.

exposes the following contract: *[loadPaper -> appendChildPaper]*. This contract states that each *loadPaper* event will cause an *appendChildPaper* to be signalled. It can be understood as follows: if the pre-condition *"loadPaper is raised"* is satisfied then**,** the postcondition *"the mod script will raise an appendChildPaper event"* will be ensured. Additionally, it should be noted that *Processing Events* are *HTML*-typed. This provides additional assurance to the website that the generated markup (i.e. the event payload) does not disrupt the aesthetics of the website.

Contract validation. When using contracts, a supplier should not try to verify that the contract conditions are satisfied; the general idea is that code should "fail hard", with contract verification being the safety net ([Wik10a]). Unit testing can be envisaged for mods, to check that a mod

meets its contract assuming its subcontractors meet theirs (i.e. the website generated appropriate *Publishing Events*). However, unit-testing design is time consuming and requires knowledge about how to obtain full coverage of test cases. This puts an extra burden on the mod script developer. On the other side, it is in the own interest of the website to thoroughly validate the mod scripts.

On these grounds, our approach leaves to the platform (i.e. the website) the duty of defining the test units. Mods need first to be verified against these test units. Only if test passes, the mod script is publicized, and liable to be installed.

Test cases generation. It is up to the website to provide test cases out of modding contracts. These contracts are based on conceptual events. For each concept, the type and domain values of each property are considered along the description in the *Modding Interface*. Specifically, two techniques are used: *Equivalence Class Partitioning*[5] and *Boundary Value Analysis*[6] ([NT08]). Finally, a pairwise testing technique is applied to warranty that each possible combination of values for every set of input variables is covered by at least one test case. As an example, the 7 properties of the concept *Paper* accounts for 93 tests.

JsUnit is a testing framework to test traditional *JavaScript* code ([Sch01]). Figure 4.5 shows this framework input page. The page prompts for an HTML page that contains a *TestSuite* (i.e. the test units to be tested). *JsUnit* enacts this page which results in a set of assertions. Each test

[5]In Equivalence Class Partitioning, the input domain is split into a finite number of subdomains where each subdomain is known as an equivalence class. This technique supposes that when one candidate of the partition class is verified then all the elements of the partition are tested. Different partition classes are created for each property depending on the type (i.e. Integer, String, Boolean) or domain (i.e. Enumerated, Ranged value) restriction.

[6]In Boundary Value Analysis, fails are located in the boundaries. This technique selects test data near the boundary of a data types and the boundaries of the equivalence classes created previously. This technique is an extension and refinement of the equivalence class partitioning technique. The test data are obtained from the boundaries of the equivalence classes depending on characteristic of the partition. As an example, if a equivalence class is a range of values (i.e. month [1..12]) then the limits are selected as a data test cases [i.e. 0,1,2,11,12,13]

accounts for an assertion. Assertions indicate either the failure or success caused by validating the tests. The outcome is shown at the bottom of Figure 4.5 for the *dblpFigures* mod. In this case, three test cases have made to fail the mod. Specifically, the message indicates that the type of the payload of the generated *Processing Event* is incorrect.

JsUnit just provides the framework while the real meat is the *TestSuite* page. It is up to the website to generate a *TestSuite* page for each mod. A *TestSuite* generates the events from the battery of test data, runs the mod for each test, and reports the result to *JsUnit*. Both the test battery and the *TestSuite* can be automatically generated from the *Modding Interface* and the *Modding Contract* so no much burden is caused to the web master.

Once the mod script is developed and tested, it is time to share among the customers of the website. In the next section, script sharing through the website is addressed.

4.9 Script Advertising

So far, script finding is achieved through script repositories. *Userscripts.org* is a case in point. These repositories act as yellow pages which offer general information about scripts: a brief description from the author, comments from the users, marks, the script code and the like. In most cases, this information is insufficient to fully understand the script behaviour, and the user resorts to install the script to see what the script looks like. This difficulty in both finding and understanding the script purpose is detrimental for the user but also for both the script programmer and the website which has lost an opportunity to engage a customer even further.

The website should take a more active role in publicizing community-provided components (i.e. mod scripts). So far, websites provide no indication about the scripts available to augment the base experience. However, it is for the benefit of the website to expose those augmented experiences to its customers. More to the point, if this additional

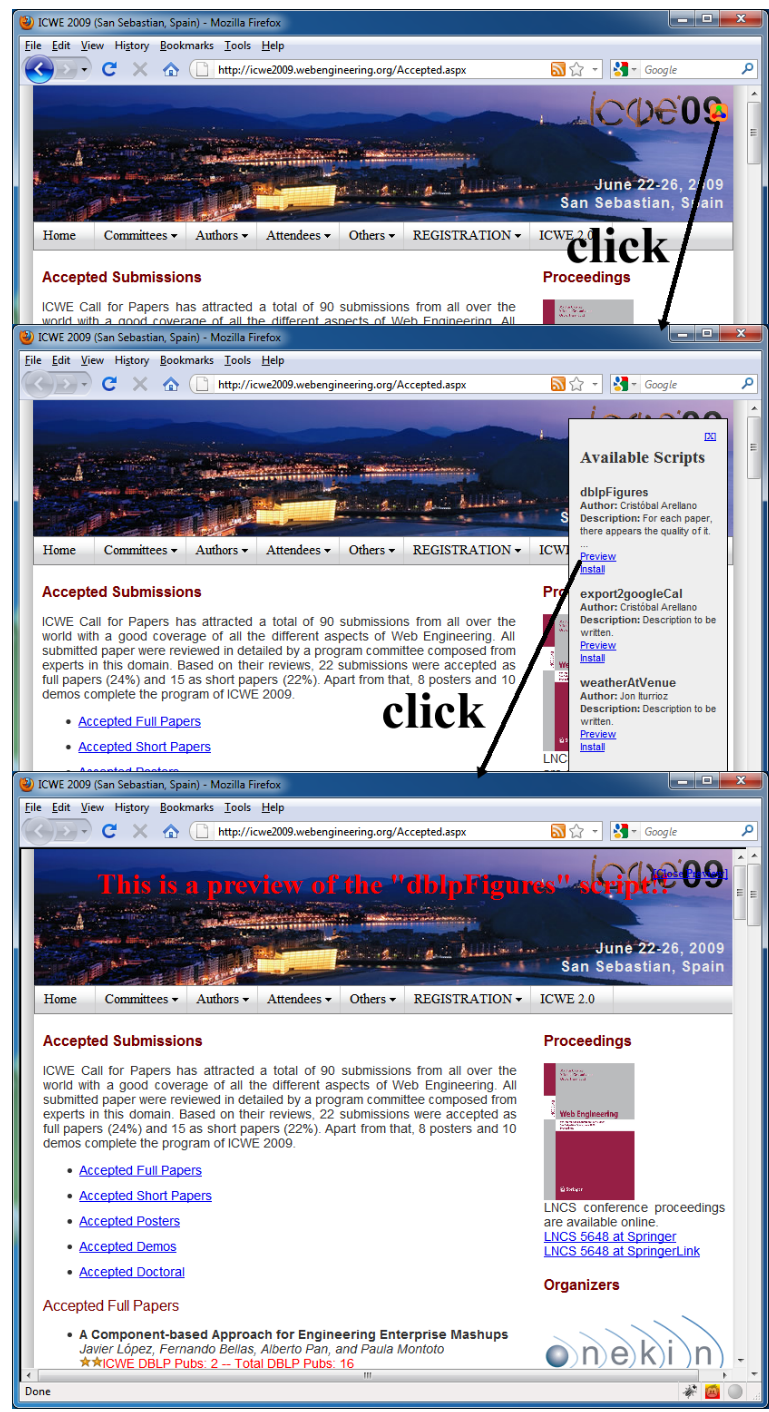

Figure 4.6: Advertising user scripts through the web site.

functionality implies no extra cost.

We introduce a second mode for websites to operate: the **preview** mode. In the traditional mode, users get the base experience, i.e. the website recovers the base pages. By contrast, the preview mode permits to visualize the core functionality but now augmented with selected mods. No additional installation is required.

The first step is to make the user aware that the website offers some augmented experiences (i.e. mod scripts). This situation is similar to advertise the existence of *RSS* channels. Current browsers are able to detect a special meta in the *HTML* heading that causes the *RSS* icon to be displayed in the menu bar. In this way, *RSS* channels are surfaced while browsing. This could have been a reasonable solution to publicize mod channels except that browsers do not recognize the script meta in the *HTML* headings. Therefore, the website itself should provide some rendering means to make users aware of this service. After all, *RSS* icons are still visible in most pages offering this service. Figure 4.6 shows the sample page but now a mod icon is displayed on the right side. By clicking on this icon, a menu bar is worked out that lists the distinct mod scripts available for this page. Besides the name, each mod script includes a brief description, one link to preview, and another link to install. Clicking on the preview causes the page to be refreshed. Now the base experience is augmented with the mod script functionality. Notice that this functionality is enacted by an event. If the event is load then, the augmented experience will be readily visible as soon as the page is refreshed. If the event is mouseover then, the augmented experience will be visible as the user pass over the appropriate page region. Figure 4.6 shows the sample page but now in preview mode for the *dblpFigures* mod.

In this way, end users can easy and safely try the augmented experience. No need to look through script repositories. Neither plug-ins nor additional installations are required. Preview pages are constructed upon base pages with additional scripting à la AJAX .

Preview pages are constructed upon base page on the fly. Specifically,

```
1 var params = ... //EXTRACT PARAMETERS FROM THE URL
2 if (params['preview']=='1'){
3  var weaver=document.createElement("script");
4  weaver.src="http://modding.icwe2009.webengineering.org/common/weaver.js";
5  document.getElementsByTagName('head')[0].appendChild(weaver);
6
7  var engine="http://userscripts.org/scripts/source/60793.user.js";
8  var script=params['script'];
9
10 weaver.addEventListener("load",function(){
11   weaverService.injectScript(engine);
12   weaverService.injectScript(script);},
13  true);
14 }
```

Figure 4.7: Code that executes the preview mode in the client.

on requesting a preview-page, an iframe is added to the raw page. This frame holds three scripts to be run at the client: the *weaver* script, the *Engine Script* (see appendix A) and the mod script whose preview has been requested. That is, it captures UI events, maps UI events into conceptual events, raises *Publishing Events* and captures *Processing Events*.

Figure 4.7 outlines the code for the *dblpFigures* preview page[7]. First, the presence of the url's 'preview' parameter is checked at line 2. If the parameter is present, then the code of the *weaver* is requested to be loaded (lines 3-5). Once the *weaver* is loaded(line 10), *Engine Script* is downloaded and enacted (line 11). Finally, the mod is executed in the same way (line 12), but its url is extracted from the url's 'script' parameter.

All of the architecture is based on the premise that offering useful *mods* to the customers of a website will increase their satisfaction. However offering malicious *mods* will cause the opposite effect, it will drive off to the customers. In the next section, the previous issue is faced.

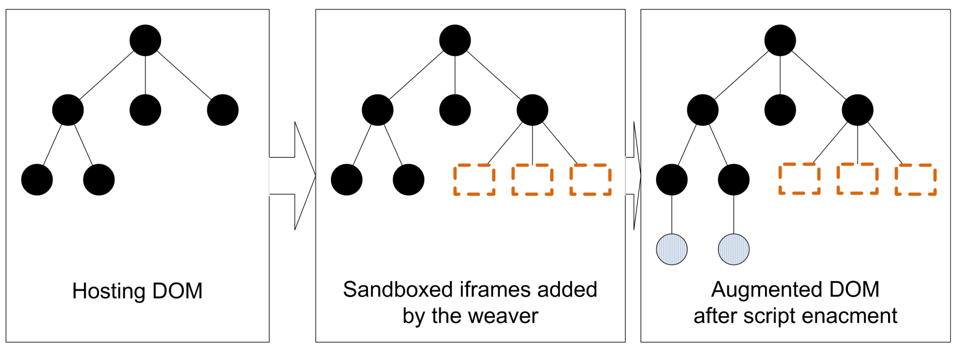

Figure 4.8: Augmentation at run-time: *DOM tree* evolution.

4.10 Script Sandboxing

Previous section advocates for websites to take a more active role in publicizing mods. Website customers are the final beneficiaries. The base experience can now be augmented through mod scripts designed by users for users. Customer loyalty, engagement and satisfaction are among the benefits for the website. However, these benefits are not without risk.

Good mod scripts boost satisfaction. But malicious mod scripts can have the opposite effect. By publicizing community-provided mods, the website's reputation might be at risk. From a user perspective, the responsibility of mod script malfunction tends to be handed over from the scripter to the website. This is particularly so for extranets where scripters might not have any contractual relationship with the website. This situation changes for intranets where scripters are employees of the organization, and hence, known by the other employees. Anyway, mechanisms are needed to shield users from malicious mod scripts.

Figure 4.8 outlines the runtime evolution of a document with embedded scripts. On loading, the document becomes a *DOM tree*. Initially, *DOM nodes* stand for the raw content of the page. Additionally, some nodes contain "cells" (denoted as doted-lined rectangles in Figure 4.8). A cell is realized as either an *HMTLDivElement* (i.e. *<div> HTML tag*) or an

[7]Being client-based scripting, the full code can be obtained through the browser by looking at the "source code" when the preview page is being rendered.

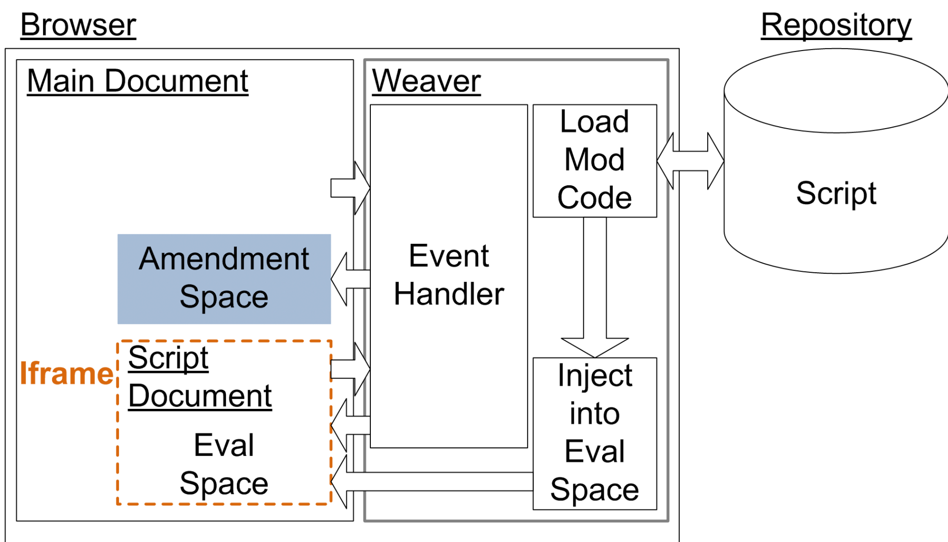

Figure 4.9: The *Modding-Interface* Architecture.

HTMLIFrameElement (i.e. *<iframe> HTML tag*) element that holds the script. Enacting the script can result on augmenting the *DOM tree* (denoted as a dot-filled circle in Figure 4.8). This figure illustrates the existence of two spaces: "the eval space" where the script is enacted (doted-lined rectangles), and "the amendment space" where the script markup is placed. In widget-oriented architectures both spaces coincides.

The *Modding-Interface Architecture (MI Architecture)* (see Figure 4.9) clearly distinguishes between the amendment space and the eval space. The amendment space is contained within the hosting document. The eval space is placed within an *"iframe jail"*. The eval space is sandboxed from the hosting document so that access is not permitted. The novelty comes from the communication model. A publish/subscribe communication model regulates the interaction between the eval space and the amendment space. A *Modding Interface* describes the messages permitted between these two spaces. Being event-driven, a *weaver* regulates publish/subscribe messages. However, and unlike *Greasemonkey*-like approaches, now the *weaver* is part of the hosting application itself. No browser plugin is required.

Therefore, engineering a web application for augmentation requires (1) a *Modding Interface* as a means to preserve application integrity and, (2) a generic *weaver* that mediates between the amendment space and the eval space. The *Modding Interface* was previously introduced in the Section 4.5. Next subsection addresses the weaver topic.

A Weaver for Augmentation Scripts

The *weaver* mediates between the main document and the script document (see Figure 4.9). Specifically, the *weaver*'s duties include (1) loading the augmentation scripts for the current user, and (2), managing *Conceptual Events*. This section outlines the implementation of these functions. The code has been tested for *Google's Chrome*, using extensively *HTLM5* new features [W3C08].

Loading Augmentation Scripts.

Customers of the web application have previously registered their interests in some augmentation scripts. These preferences are kept locally through a *localStorage* variable at the browser: *augmentationConfiguration*. Scripts are kept at the server. Figure 4.10 lists the *weaver*'s code that loads the scripts.

On loading the web application, the *weaver*'s first duty is to load the script identifiers kept at *augmentationConfiguration* (lines 2-3). For each script, the *weaver* creates an *iframe* (line 7-11). An *iframe* holds a generic document (*src* attribute) that is parameterized with the identifier of the script at hand. This document has no rendering counterpart (i.e. *"display:none"*). *Iframes* are sandboxed. When the *iframe* is added to the page (line 12), the script is downloaded and evaluated. Being sandboxed, the script cannot access the hosting page (i.e. the script cannot subscribe to UI events from the main document). Interactions are restricted to occur through a channel (line 14-17). A message channel is an *HTML5* object that enables the direct communication of independent pieces of code

```
1  // Load configuration
2  var installedScripts=localStorage.getItem("augmentationConfiguration");
3  installedScripts=installedScripts?JSON.parse(installedScripts):[];
4  // Load scripts
5  for(script in installedScripts){
6   // Create of sandbox
7   var iframe=document.createElement("iframe");
8   iframe.src="http://icwe2009.webengineering.org/mod_document.html?"+
9              installedScripts[script].id;
10  iframe.style.setProperty("display","none","important");
11  iframe.sandbox="allow-scripts";
12  document.body.appendChild(iframe);
13  // Initialize communication
14  var channel = new MessageChannel();
15  iframe.addEventListener("load",function(){
16     iframe.contentWindow.postMessage("initChannel",[channel.port2],
17     "http://icwe2009.webengineering.org/"); },true);
18   ...
```

Figure 4.10: *Weaver*'s code: loading augmentation scripts.

(e.g. running in different browsing contexts). This interaction follows a publish/subscribe pattern based on *Conceptual Events*.

Managing Conceptual Events.

When the *iframe* space is initialized, the main document and the script document are ready for exchanging *Conceptual Events*. However, these *Conceptual Events* are to be produced/handled by the *weaver*. The *weaver* has two main duties: raising *Publishing Events* in the main document, and handling *Processing Events* as signalled by *script* documents.

The process goes as follows. The UI event (e.g. loading a page) is first notified to the *weaver*. The *weaver* constructs and raises the *Conceptual Event* (e.g. *loadPaper*) along the indications of the *Modding Interface*. *Conceptual Events* are captured by the *script* that recovers the event payload, constructs an *HTML fragment*, and dispatches the appropriate *Processing Event* (e.g. *appendChildPaper*). This *Processing Event* is then de-constructed in terms of UI operations by the *weaver* according to the indications of the *Modding Interface*. These UI operations causes the main document (the page you see) to be augmented.

```
<head>
 <link rel="moddingInterface" href="/modding_interface.owl" />
 <link rel="transformation" href="/extractor.xsl" />
 <script type="text/javascript" src="/weaver.js"></script>
 ...
```

Figure 4.11: Augmentation-enabled page: meta-data about the *Modding Interface*.

4.11 Discussion

In this section, the four requirements, namely, affordability, resilience, scalability and security are revisited and evaluated in terms of the proposed solution.

4.11.1 Affordability

In a crowdsourcing setting, the viability of an approach heavily rests on causing minimal disturbance to the involved parties: web application programmers and script programmers. As for the former, the *MI Architecture* imposes almost no disruption. Augmentation-enabled *HTML* pages differ from traditional pages in that they keep three links: two to the *MI* files, another to the *weaver* script (see Figure 4.11). Apart from that, these pages do not differ from "traditional pages".

From a script-programmer perspective, *MI* implies notification/publication to be based on *Conceptual Events* rather than *DOM events*. Otherwise, native *JavaScript* mechanisms are used to handle *Conceptual Events* with no variations w.r.t. traditional script development. From the start of this work, we have been very conscious about reducing the hurdles for offering contributions. Next paragraphs provide evidence that programming on top of a *Modding Interface,* causes minimal deviation from traditional practices.

Notification of Processing Events. *JavaScript* follows an event-based approach where handlers can be associated with *DOM-based events*. Operations are available for creation of event occurrences (e.g.

createEvent("MouseEvents")), assigning the payload to an occurrence (e.g. *initMouseEvent("eventInstance", "eventParameters")*), or raising the event manually (e.g. *dispatchEvent(eventOccurrence)*). Raising of *Conceptual Events* uses these standard *JavaScript* operations. Back to our running example, a *dblpFiguresPanel* (i.e. an *HTML fragment*) is to be injected as a child of a *Paper*. Figure 4.2 (right side) show the code along the following pattern: *createEvent* (lines 28-29), obtain concept (line 21), *initProcessingEvents* (line 30-31), *dispatchEvent* on this concept (line 32). This is standard *JavaScript* code. The only difference with traditional scripting is that now the injection point is not a *DOM node* but the current concept. This current concept is to be obtained through a *Publishing Event*.

Subscription to Publishing Events. *JavaScript* achieves event subscription by registering a handler through the *addEventListener* method. Subscription to *Conceptual Events* is accomplished in the very same way: associating a handler. For instance, instruction (line 38 in Figure 4.2 (right side)) *"doc.addEventListener("loadPaper", init, true)"* adds a handler to the *loadPaper* event, i.e. occurrences of *loadPaper* will trigger the *init()* function. The difference rests on handlers being associated to the whole document (i.e. variable *doc*) rather than to *DOM nodes* (e.g. a *checkbox*). This highlights the fact of events being raised by acting on *Papers* rather than on *DOM nodes* (i.e. the circumstantial representation of these *Papers*).

4.11.2 Resilience

Mods should be resilient to website upgrades. This is the rationale of the *Modding Interface*: changes in the content or layout of a website should not impact the *mod*. Even if a concept (e.g. *Paper*) is no longer rendered, the *mod* will still raise the event, but the website will not take care for it. No dangling references come up. The *mod* becomes redundant but not faulty. And vice versa, new webpages can be introduced where *Paper* 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 webpage. 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.

4.11.3 Scalability

All our measurements are realized in *Windows 7 x64* running on *Intel Core2 Duo 2.20 GHz CPU* with *4GB of memory*. The experiments have been realized with a domestic *6Mbps WIFI LAN* bandwidth.

Loading time. The *Greasemonkey* architecture keeps scripts at the client. So, no loading penalty at the time the script is enacted. By contrast, our approach makes scripts a valuable asset of the web application which becomes a partner on disseminating these resources among its user base. Therefore, the *MI Architecture* maintains scripts at the server as site resources. When application pages are loaded, so are the appropriate scripts (as any other page resource such as associated images). Compared with *Greasemonkey*, this certainly imposes an overhead. However, script files tend not to be very large, and its cost is similar to loading a *"jpg"* thumbnail file. Additionally, the *weaver* and *Modding Interface* file are loaded on accessing the first page. The size of the *weaver* file is 3.8kb (no obfuscated) which approximately accounts for a 100 millisecond delay (less if the *weaver* is cached by the browser). The size of the *Modding Interface* for a given page is similar to a script. On the upside, this approach frees users from installing any plugin (as it is the case for *Greasemonkey*).

Enactment time. Script enactment takes place at the client (no server impact). *Greasemonkey* scripts act upon *DOM events*. By contrast, interface-aware scripts rest on *Conceptual Events*. This imposes an indirection: *Conceptual Events* need first to be (de)constructed from *DOM events* and send over the channels that connect the script space with the hosting application space. A first experiment has been conducted for the *dblpFigures* sample realized as both a *Greasemonkey* script and

an interface-aware script. The results show that the indirection and communication accounts for a delay of 30 and 2 milliseconds, respectively, when compared with the *Greasemonkey* alternative (i.e. acting directly upon *DOM events*).

4.11.4 Security

Both redirection to phishing pages or stealing sensitive data are avoided by running the script inside an *"iframe jail"*. On the other side, we can prevent port scanning and history sniffing by using the same approach as *Google Caja*: a *monkey patch* [Wik10b]. *Monkey patch* is a way to extent/modify runtime code in dynamic languages. This technique can be applied to dynamically replace/extend script functions liable to content malware with others that block such malware. Finally, browser blocking can be alleviated as in *MS' Web Sandbox*, i.e. using a QoS Layer [Micb]: a wrapper-like mechanism that imposes some limits on the consumption of shared resources. Exceeding these thresholds (e.g. CPU consumption) makes the script be blocked.

4.12 Conclusions

Web-based open ecosystems are predominantly API-based and service-oriented. As an alternative, this chapter introduces a front-end, script-based approach. Fostering a win-win relationship between website owners and website users, substantiates the efforts for websites to become scripting platforms. From the website owner's viewpoint, the *Modding Interface* realizes a controlled setting for modding that can bring cost reduction, increased innovation, and quicker development time. From the scripter's perspective, the *Modding Interface* reduces the freedom but increases change resilience, and eases coding. The approach benefits web applications that now can be safely augmented. So does for script contributors that now achieve greater visibility by having their scripts

uploaded at the hosting application.

In this chapter, we focus on the technical feasibility of *HOP*, identifying and addressing four main issues. First, the insufficient decoupling between user scripts and the underlying website, which is handled by introducing *Modding Interfaces*, increases the resilience of the solution and serves as a security barrier between website and *mod* scripts. The event-based nature of *Modding Interface* mimics the event-based nature of the *DOM* scripting in order to improve the affordability of the solution. Second, *Modding Contracts* are proposed for the website to automatically test the mod scripts. We believe that testing the scripts, will improve the soundness of community-provided scripts. Third, the preview mode is introduced to expose external contributions to the website's users. The user does not longer need to search for additional functionality but this is offered by the website with no extra cost for the user. Fourth, the regulation of inflow/outflow communication between the website and the scripts using the *Modding Interface* and a weaver that enables the safe execution of scripts. Combining *"iframe jails"* and "modding-interface" *HTML5 channels,* the security issues, where malware can cause important damages on end users hence in the reputation of hosting applications, are blocked.

Chapter 5

Client-Side Open Personalization

5.1 Introduction

Web Augmentation is to the web what *Augmented Reality* is to the physical world: layering relevant content/layout/navigation over the existing web to customize the user experience. This is achieved through *JavaScript (JS)* using browser weavers (e.g. *Greasemonkey*). To date, over 43 million of downloads of *Greasemonkey* scripts ground the vitality of this movement. However, *Web Augmentation* is hindered by being programming intensive and prone to malware. This prevents end users from participating as both producers and consumers of scripts: producers need to know *JS*, consumers need to trust *JS*. This paper aims at promoting end user participation in both roles. The vision is for end users to *prosume* scripts as easily as they currently *prosume* their pictures or videos. Encouraging production requires more "natural" and abstract constructs. Promoting consumption calls for augmentation scripts to be easier to understand, share and trust upon. To this end, we explore the use of *Domain-Specific*

Languages (DSLs) by introducing *Sticklet*. *Sticklet* is an internal *DSL* on *JS*, where *JS* generality is reduced for the sake of learnability and trustworthiness. Specifically, *Web Augmentation* is conceived as fixing in existing websites (i.e. *the wall*) *HTML* fragments extracted from other sites or web services (i.e. *the stickers*). *Sticklet* targets hobby programmers as producers, and computer literates as consumers. From a producer perspective, benefits are three-fold. As a restricted grammar on top of *JS*, *Sticklet* expressions are domain-oriented and more declarative than their *JS* counterparts, hence promoting production and understanding. As syntactically correct *JS* expressions, *Sticklet* scripts can be installed as traditional scripts and hence, programmers can keep using existing *JS* tools. As declarative expressions, they are easier to understand (and so to trust upon) and amenable for optimization where the *DSL* engine can decide the most efficient way to carry out an action. From a consumer perspective, domain specificity also permits to customize the installation/enactment/sharing of *Sticklet* expressions (as compared with JS tools) to address the shortage of time and skills of the target audience. Preliminary evaluations indicate that 77% of the subjects were able to develop new *Sticklet* scripts in less than thirty minutes while 84% were able to consume those scripts in less than ten minutes. *Sticklet* is available to download as a *Mozilla* add-on.

This chapter is organized as follows. Section 5.2 motivates *Client-Side Open Personalization* using some *Web Augmentation* examples. From the previous examples, in Section 5.3, the requirements of the solution are enunciated from the attention available/required viewpoint. Sections 5.4, 5.5 and 5.6 introduce *Sticklet*, an internal *DSL* on *JavaScript*, where the generality of *JS* is reduced to meet the requirements. Finally, in section 5.8, the requirements are revised from the proposed solution viewpoint. Conclusions end the chapter.

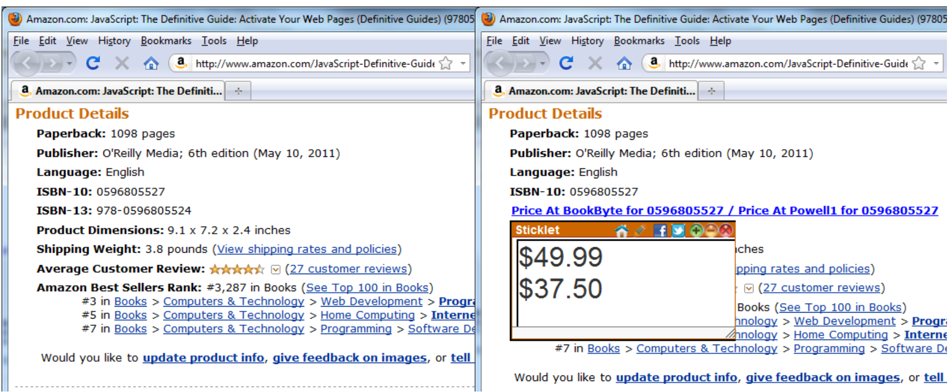

Figure 5.1: *Amazon* before and after the *BookBurro* augmentation.

5.2 Motivating Scenario and Research Question

Web Augmentation can support a broad range of situational scenarios:

- On browsing an online journal (*e.g. USA Today*), you can be interested in the coverage that a given headline receives in another online newspaper, e.g. *The NY Times*. Skipping to the *TNYT* and searching for a related headline could be too cumbersome to do on a routine basis. Rather, you would like the *USA Today* website to be augmented with a button placed by the *USA Today* headline that directly pops up the summary at *TNYT* for this headline.

- When rendering a book at *Amazon*, it could be useful to know the prices/comments for this book at other online bookshops.

- On weighting a job post at *www.monster.com*, it could be of interest to supplement *monster* data with information about the range of wages and conditions of similar jobs as found in other web sites (e.g. *jobs.trovit.co.uk*).

These examples illustrate **short-term situational scenarios of end-user Web Augmentation** (hereafter just *"Web Augmentation"*). The purpose

is to make the web more responsive to the unique and individual needs of each user. From this perspective, augmentation shares aims with *Web Personalization* [RSG01]. The difference stems from *who* sets the personalization. In the case of personalization, the application designer is in charge while users are passive consumers. The issue is that some personalizations might be of interest only for a small number of users (hence, lacking the scale that makes the personalization payoff), fall outside the business model of the web application (e.g. *Amazon* lacks the interest in putting up comparative prices from other online bookshops) or be difficult to foresee by the application designer. This discrepancy between what application developers can build, and what individual end-users really need can be addressed with *End-User Development* [RI06].

Using special weavers, third-party *JavaScript* code can make on-the-fly changes to the currently loaded web page. Weavers are available for *Firefox* (e.g. *Greasemonkey*), *Internet Explorer* (e.g. *IE7Pro* or *Turnabout*), *Safari* (e.g. *SIMBL + GreaseKit*), and natively supported in *Opera* and *Google Chrome*. The running examples for this paper were tested for *Greasemonkey (GM)* [LBS05] although the solution is browser agnostic. As an example, consider a popular script: *BookBurro*[1]. This script embeds price comparison in *Amazon* pages. Figure 5.1 shows the outcome before and after applying the script that injects the *BookBurroPanel*. This is achieved at the browser through the weaver. Weavers permit scripts to act upon web pages at runtime. Pages are realized as *DOM* trees[2]. The script is triggered by User Interface events (UI events) on this *DOM* tree (e.g. *load*, *click*). Event payloads provide the data to feed script handlers which, in turn, update the *DOM* tree. The script is outlined in Figure 5.2. The process goes as follows:

- interacting with a page triggers UI events (e.g. *load*),

[1] *BookBurro* is available at *http://userscripts.org/scripts/source/1859.user.js*.

[2] The Document Object Model (*DOM*) is a platform- and language-independent standard object model for representing *HTML* or *XML* documents as well as an Application Programming Interface (API) for querying, traversing and manipulating such documents.

- the script *reacts* to this event by triggering a handler (lines 6-39). The association between an event and a handler (a.k.a. event listener) is achieved through the *addEventListener* function (line 6),
- a handler can access any node of the page (using *DOM* functions such as *document.evaluate* in lines 9-10), and create *HTML* fragments (e.g. line 21),
- a handler can also *change* the *DOM* structure at will by injecting *HTML* fragments (e.g. the *BookBurroPanel*). In the example, the output is injected at a point identified by an *XPath* expression on the underlying *DOM* structure (i.e. the injection point). *DOM* functions are used for this purpose (e.g. *appendChild* in line 23 and 36),
- this script is associated with a URL pattern that denotes the pages to which the script applies. This is specified through the *@include* annotation (line 3).

From a producer perspective, this approach incurs in an important drawback: scripts are vulnerable to page changes. Back to our sample case, if the *Amazon* website is upgraded, all "the screen scrapping" can fall apart. For instance, *BookBurro* first retrieves the book's *ISBN* from the current page, and next, injects the *BookBurroPanel* at a certain location. This is normally achieved through *XPath* expressions (line 9). If *Amazon* pages are changed then, *BookBurro's XPath* expressions could no longer recover/identify the right *DOM* node. Therefore, *GM* scripts are specially prone to maintenance. Besides their own maintenance, scripts are affected by the maintenance of the hosting website. The problem is that websites are reckoned to evolve frequently.

From a consumer perspective, a main concern is *script collision*, i.e. the simultaneous access to the same *DOM* node by two different scripts. The very same web page can be subject to different augmentations. *Amazon* is a case in point. At the time of this writing, 268 scripts are reported to be available for *Amazon* at *userscripts*. If you are a regular

```
1  // ==UserScript==
2  // ...
3  // @include        http://www.amazon.com/*
4  // ...
5  // ==/UserScript==
6  window.addEventListener("load", function(){    //(STICKLET)
7   // Script scope: structural condition + content condition on the current document
8   // Checking structural condition (STICKLET'S SELECTBRICK)
9   var isbns=document.evaluate("//li[contains(b/text(),'ISBN-10')]",
10                               document, null, XPathResult.ANY_TYPE, null);
11  var nodes=[];
12  for(var elem=isbns.iterateNext();elem!=null;elem=isbns.iterateNext())
13                                            {nodes[nodes.length]=elem;}
14  // For each document portion fulfilling the structural condition
15  for(var i=0;nodes.length>i;i++){
16   var isbnNode=nodes[i];
17   // Checking content condition (STICKLET'S EXTRACTCONTENT,AS)
18   var isbn=isbnNode.innerHTML.match(new RegExp("ISBN-10:</b> (\\d{10})"))[1];
19   if(isbn!=null){
20    // Adding the augmentation lever (STICKLET'S INLAYLEVER,AT)
21    var link=document.createElement("a");
22    link.innerHTML="Price At BookByte for "+isbn;
23    isbnNode.appendChild(link);
24    (function(_link,_ad,_isbn){
25    // Listen to the user interaction (STICKLET'S ONTRIGGERINGLEVERBY)
26    _link.addEventListener("click",function(){
27     //Invoking augmentation service (STICKLET'S LOADNOTE)
28     GM_xmlhttpRequest({
29      method: "GET",
30      url: "http://www.bookbyte.com/product.aspx?isbn="+_isbn,
31      onload: function(response) {
32       if(response.status!=200){alert("Error: "+response.statusText);return;}
33       //Rendering service response, encapsulated in "visualize" call (STICKLET'S STICKNOTE)
34       var bookBurroPanel=visualize(response.responseText,
35                       "//span[@id='ctl00_ContentPlaceHolder1_lblBestNew']","(.*)");
36       _ad.appendChild(bookBurroPanel);
37      },
38      onerror:function(response){alert("Server Not Exist");}});},true);
39    }(link,isbnNode,isbn));}}},true);
```

Figure 5.2: *BookBurro* in *JavaScript* (partial view).

Amazon visitor, it is likely you have several scripts installed. These scripts will be enacted simultaneously when you visit *Amazon*. It is important to notice that script execution is not in parallel but in sequence, i.e. scripts are launched in the order in which they were installed. This implies that the first script acts on the original *DOM* tree, the second script consults the *DOM* tree but once updated by the first script, and so on. The problem is that programmers develop scripts from the original *DOM*, being unaware of changes conducted by other companion scripts. This can end up in a real nightmare where code developed by different authors with different aims, is mixed up together with unforeseen results. Even worse, the final *DOM* tree can even be dependent on the order in which scripts are enacted! The

larger the set of (companion) scripts, the higher the likelihood of clashes. This problem, coupled with the fact that the number of scripts is steadily growing, will likely lead to an increase in the number of scripts in each *user* installation, and hence, in the likelihood of collisions.

The bottom line is that the expressiveness brought by a general-programming language such as *JavaScript* comes at the price of intensive development, and, what is most important, maintenance. Consumption also suffers from this freedom. Even fully-tested scripts (e.g. the *Skype* button) can collide when enacted simultaneously with scripts that access the same *DOM* regions. The problem is that these errors are detected (and suffered) by consumers with little help from producers who can hardly foresee the context in which their scripts are to be run. This potentially high cost of development, maintenance and consumption, compromises the “end-userness” of *JavaScript* for *Web Augmentation*. Next section trades *JavaScript* expressiveness for maintainability and reliability.

Previous scenario serves to illustrate the research question:

> ***How can we help end-users to create, share and consume functionality that augments the content of websites?***

Next section describes the requirements of the solution in terms of the theory of *“Attention Investment”*.

5.3 Requirements

The theory of *"Attention Investment"* has been proposed as a basis for the design of *End-User Development* systems [BG99, Bla02]. This theory describes users’ decisions about how to allocate their attention in problem-solving as investments. Drawing on these insights, we arrange requirements based on two aspects of the target scenario: users and tasks. The user is characterised in terms of **the attention shortage** to perform the task. The task is described in terms of **the attention required** to accomplish the task (see Figure 5.3). Different tools can

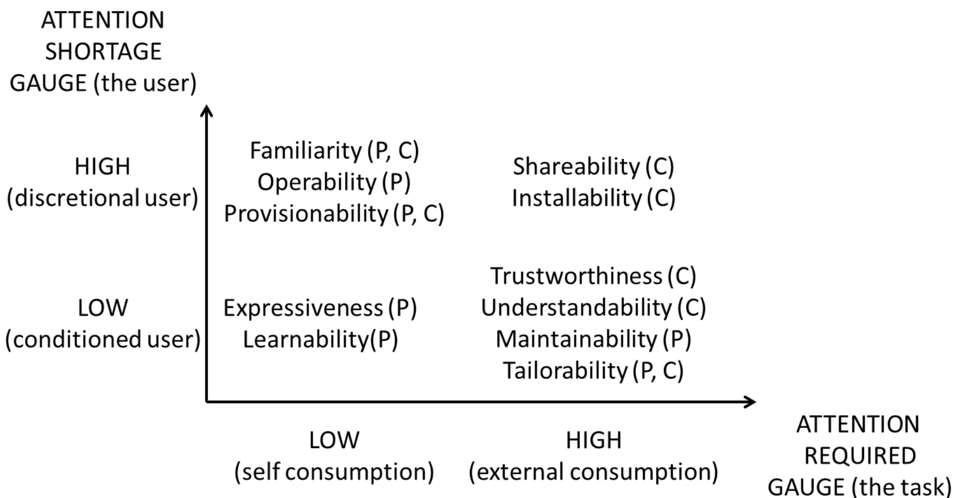

Figure 5.3: *Sticklet* Design Drivers. Criteria are qualified by “P” or “C” based on their biased towards producers or consumers, respectively.

be designed that weight these two criteria differently. Although these dimensions go along a continuum, we provide just two discrete values for each dimension. The "attention-shortage gauge" distinguishes between **conditioned prosumers** *versus* **discretional prosumers**. The former conducts augmentation within an organization, and is commonly job-oriented. This results in the attention-shortage gauge pointing to "low". By contrast, discretional prosumers conduct augmentation in a less pressing environment, and usually for self reward. This leads the attention-shortage gauge to point to "high".

As for the task, it is characterised in terms of the perceived difficulty. In this sense, we distinguish between scripts intended for **self-consumption** (attention-required gauge pointing to "low") or scripts for **external consumption** (attention-required gauge pointing to "high"). This is aligned with the work of [KAB+11] where producers’ intents (i.e. the envisaged audience of the scripts) determine to what extent producers “consider concerns such as reliability, reuse, and maintainability and the extent to which they engage in activities that reinforce these qualities, such as testing, verification, and debugging” [KAB+11]. The difference stems

from the additional attention required when your code is to be consumed by others.

These two dimensions help to arrange requirements for augmentation tools along the so-identified quadrants. The more stringent the demands in terms of the complexity of the task ("attention required") or the effort available to accomplish the task ("attention shortage"), the larger the set of attributes the tool should cater for. The bottom left quadrant (i.e. low attention required, low attention shortage) represents production by programmers that use augmentation to speed up some personal routine tasks. One of the main challenges in this scenario is to find a balance between **expressiveness** and **learnability**. More complex languages can address a wider range of problems but impose an increasing learning burden on users.

As we move rightwise, production targets a wider audience. Besides producers, this quadrant introduces consumers as first-class stakeholders. Producers will look for support in testing, verification, and debugging of their scripts since **trustworthiness** (e.g. reliability) and **maintainability** become main concerns. From the consumer perspective, they surely value trustworthiness but also **understandability** or **tailorability** (i.e. permitting the consumers to adapt the script by themselves [DLL06]). Some *DSLs* fall in this quadrant. Here, it is not expected for end-users to produce the *DSL* expressions on their own but just to read the code, understand what it means, and talk to programmers directly about necessary modifications [Fow09]. In this setting, *DSLs* are not a substitute for programmers but a way to increase their productivity while improving the reliability and understandability of software.

If we move upwards, we confront the shortage of attention. If attention is scarce, new requirements come into play. First, the operation of the whole tool should be intuitive (i.e. **operability**). If possible, tool design should capitalize on whatever aspect the target audience is familiarized with so that users can reapply what they already know (i.e. **familiarity**). Moreover, production tends to be less systematic where users commonly

resort to evolutionary and exploratory prototyping (i.e. **provisionability**) [GBC+06]. In addition, producers might be motivated by the possibility of exhibiting the final product to others as a demonstration of skill and technical mastery. Web2.0 scenarios make consumption a main incentive for production. This moves us to the upper right quadrant, i.e. the promotion of consumption in discretional scenarios. Two additional aspects emerge: **installability** (i.e. the quality of requiring minimum installation burden) and **shareability** (i.e. facilitating script sharing to fuel consumption).

By no means, we claim this is a complete list of requirements. Rather, this design space serves to frame the main design criteria considered during the *Sticklet* implementation. Design choices are subject to tradeoffs between factors that will value some attributes while penalizing others. For instance, *JavaScript* tools mainly value expressiveness at the cost of penalizing understandability and reliability. On the other hand, *Visual Programming Tools* and *API*-based approaches favour easy production by trading expressiveness for learnability. Unfortunately, no matter the approach, all leave consumers to face raw *JavaScript* code.

This work sets augmentation in a Web2.0 scenario: scripts are not only for self-consumption but sharing, where sharing in turn, fuels production. This virtuous cycle is not however a free lunch. Easing consumption might imply reducing the expressiveness as well as reducing the visual aids to favour the operability and installability of the solution. There is thus no ideal tool for any augmentation situation, only designs that are more or less well suited to the activities of the people doing the augmentation. For the purpose of this paper, these people are hobby programmers as producers, and computer literates as consumers.

5.3.1 Existing Solutions

This work takes inspiration from two main areas: *End-User Development* and *Web Mashups*. Like *mashups*, *Web Augmentation* also reuses

	PLATYPUS	MASHMAKER	ACTIVETAGS
Operability (P)	++ (graphical editor for specification but no debug /tracing)	+ (text/graphical with no debug /tracing)	+ (text/graphical with no debug/tracing)
Provisionability (P,C)	not applicable (n.a)	n.a.	n.a.
Expressiveness (P)	+ (single-page customization)	+++ (mashup customization)	- (focuses on tagging sites)
Learnability (P)	+++ (graphical)	- (*JS*+*XML*+...)	+++ (graphical)
Maintainability (P)	n.a. (scripts can not be opened in the editor)	- (Widget /gadget code + GUI)	n.a.
Shareability (C)	+++ (*JS* general repository)	++ (proprietary repository)	++ (proprietary repository)
Installability (C)	+++ (1 click)	+ (1 to 6 click depending on the script)	?
Trustworthiness (C)	- (that of *JS*)	++ (built-in)	++ (no security measures)
Understandability (C)	- (that of *JS*)	- (procedural and multiple artifacts /languages)	++ (n.a)
Tailorability (P,C)	- (directly on the *JS* raw)	++ (configuration-based via widget properties)	-

Table 5.1: Web Augmentation frameworks (visual approaches).

existing web resources. Unlike *mashups*, augmentation does not aim at creating a new application but complementing an existing one. This implies that common mashup techniques for the discovery, selection or composition of web resources [CDM+11, DCBS09, DRC+12] should now be contextualized by its relationship to the targeted website. In addition, *Sticklet* regards *HTML* pages as the main resource to tap into rather than

	CHICKENFOOT	IE ACCELERATOR
Operability (P)	++ (text editor with introspection)	++ (text with no debug /tracing)
Provisionability (P,C)	++ (code increments)	n.a.
Expressiveness (P)	+++ (customization & automation)	+ (button addition)
Learnability (P)	++ (*JS* + library)	+++ (reduced *XML*)
Maintainability (P)	+ (*JS* code with some abstractions)	n.a.
Shareability (C)	+++ (*JS* general repository)	+++ (proprietary repository)
Installability (C)	+ (7 clicks)	+++ (1 click)
Trustworthiness (C)	- (that of *JS*)	++ (no security measures)
Understandability (C)	+ (library provides abstractions)	++ (*XML* document)
Tailorability (P,C)	+ (via prompts)	-

Table 5.2: Web Augmentation frameworks (textual approaches).

RSS or *API* services, as it is the most common case in mashups. At this respect, an interesting work is that of [GPS11] where the "mashup components" are wrappers upon websites, and the "composition model" is side-by-side integration. The resulting mashup application looks like a "quilt" of website windows. Notice, however, that the "quilt" is a standalone application, different from the "website patches". In brief, current mashup abstractions and composition paradigms depart from the mental model proposed for augmentation development. The notion of *"the sticker wall"* might better capture the asymmetry that exists between the website being augmented and the rest of the websphere. Consequently, existing *mashup* work is of interest for *Web Augmentation* but needs first to be tuned to these peculiarities.

The aim of simplifying *Web Augmentation* has also been addressed using a range of techniques (e.g. *Visual Programming Tools*, *APIs*, or hybrid architectures) and tackling different augmentation scenarios:

generic augmentation (i.e. augmentation that can be conducted for any website), niche augmentation (i.e. augmentation that focuses on a specific kind of sites), and opportunistic augmentation (i.e. unplanned augmentation). The rest of this section uses a representative tool to ground each approach: *Platypus* [Tur05] (*Visual Programming* approach), *Intel MashMaker* [EBG+07] (hybrid approach), *ActiveTags* [HV09] (niche augmentation), *Chickenfoot* [BWR+05] (*API* approach) and *IE Accelerator* [Mica] (opportunistic augmentation). Tables 5.1 and 5.2summarizes the insights.

Platypus is a *Visual Programming Tool* for generic augmentation. It obtains full-fledged *JavaScript* code for *Greasemonkey* using a graphical toolbar. Users directly act upon the current page through the *Platypus* toolbar, e.g. supressing banners, moving parts of the page to different locations, changing the style and format of page elements, or inserting their own *HTML* code. From a producer perspective, *Platypus* is a neat tool for its purpose: changing a web page based on the page itself. On the downside, visual tools might restrict the expressiveness to facilitate code generation (e.g. in *Platypus*, no page other than the current page can be accessed). Hence, it is not clear how a visual approach will scale up as the augmentations become more complex. The more detailed claim that visual notations avoid the need to learn a syntax appears dubious (e.g., [Gli89, GPB91]). In addition to the practical problems of real state of the screen and visual clutter, graphical programming languages suffer from being difficult to port (because of the graphics) and expensive to develop because of the high cost of building the necessary editors, compilers, and debuggers [Mye90]. From a consumer perspective, visual tools frequently behave as generators of *JavaScript* code. This hides the complexities for producers but leaves consumers with convoluted, machine-generated code, hence, obfuscating the origin of errors, or interfering with effective communication and explanation. Notice however that these tools tend to be used for self-consumption, and hence, consumption of third-party scripts is not an issue.

MashMaker illustrates a hybrid approach for generic augmentation. A distinctive aspect is that programmers and end-users asynchronously collaborate to come up with the augmentation. A *MashMaker* project encompasses three artefacts: the *data extractor* (graphically defined), the *augmentation widget* (which is separately coded in *JavaScript*), and the so-called *"mashup"* (graphically defined). The *"mashup"* links the two previous artefacts so that the *widget* is fed from the *extractor*. A library of *widgets* is made available by programmers to end users. This introduces two actors during augmentation: *widget* programmers and end-user "linkers".

ActiveTags is a visual tool that illustrates "niche augmentation". Here, the scope of the augmentation is restricted so that the system can automatically infer how to extract some data, relieving the user from this burden. *ActiveTags* limits augmentation to tagging systems (e.g. *del.icio.us*, *Flickr*, etc.) where tags are always the data to be extracted and the augmentation levers. That is, *ActiveTags* permits to associate "an augmentation service" to the appearance of a tag. By clicking on a tag, the service is invoked and the returned markup is popped up. This focus permits extractors (supported in *Sticklet* through *SelectBrick*, *ExtractContent*, *A*s) and augmentation levers (*InlayLever*, *At*, *OnTriggeringLeverBy*) to be hidden from users. This improves learnability for this niche domain. On the other side, consumer concerns are not explicitly addressed since *ActiveTags* expressions are thought for self-consumption.

Chickenfoot illustrates the *API* approach for generic augmentation. An *API* introduces some abstractions that shelters users from how these abstractions are implemented *but without leaving the hosting language*. In this way, users of *API*-leveraged languages can use *API* methods, and resort to general instructions when they require to do so (e.g. *Chickenfoot* methods can hide complex heuristics about how to extract some data based on nearby text). This results into a leaner code, easier to write and understand. Figure 5.4 provides the *BookBurro* example

```
1 // ==UserScript==
2 // ...
3 // @include        http://www.amazon.com/*
4 // ...
5 // ==/UserScript==
6 // Script scope: content condition on the current document
7 // Checking content condition (STICKLET'S EXTRACTCONTENT,AS)
8 var isbns=find(/ISBN-10: (\d{10})/);
9 for(;isbns.hasMatch;isbns = isbns.next){
10  var isbn=isbns.groups[1];
11  // Adding the augmentation lever (STICKLET'S INLAYLEVER,AT)
12  var link=document.createElement("a");
13  link.innerHTML="Price At BookByte for "+isbn;
14  insert(after(isbn),link);
15  (function(_link,_isbn){
16    // Listen to the user interaction (STICKLET'S ONTRIGGERINGLEVERBY)
17   _link.addEventListener("click",function(){
18    //Invoking augmentation service (STICKLET'S LOADNOTE)
19    var response=fetch("http://www.bookbyte.com/product.aspx?isbn="+_isbn);
20    whenLoaded(function(){
21     var error=response.document.baseURI.match(/about:neterror?e=([a-zA-Z]+)/);
22     if(error){
23       alert("Error: "+error[1]);
24     }else{
25      //Rendering service response, encapsulated in "visualize" call
26      //(STICKLET'S STICKNOTE)
27      var bookBurroPanel=visualize(response,/\$(\d+\.\d+)/);
28      insert(after(_isbn),bookBurroPanel);
29     }
30     response.close();
31     },response);
32  },true);}(link,isbn));
33 }
```

Figure 5.4: *BookBurro* using *Chickenfoot* API. Bold stands for calls to the *Chickenfoot* library.

now as a *Chickenfoot* script (bold is used for the *Chickenfoot API* calls). *Chickenfoot* pioneers content extraction from *HTML* pages based on so-called *text constraint patterns* using the *LAPIS* library [MM00]. The use of *LAPIS* (a *Java API*) in *Chickenfoot* (a *JavaScript* programming system) implies a penalty in terms of the loading of the virtual machine. This might be the reason why the latest releases of *Chickenfoot* resign from using *LAPIS*, and support an abbreviated form of content extraction which is realized as *JavaScript* functions. For instance, retrieving the *ISBN* from *Amazon* pages is expressed as *"after(text isbn-10)"* (line 8)[3]. In the

[3]Unfortunately, *Chickenfoot* heuristics do not work properly for the *Amazon* page, and the “after” call does not retrieve the *ISBN*. This can be settled by substituting the “after”

example *"text"* is an *HTML* type whereas *"ISBN-10"* is a literal. Functions *"after"* and *"before"* are available to retrieve the content of the node that follows/precedes the node identified by this expression. Therefore, *Chickenfoot* scripts are easier to develop and understand than using directly *JS*. However, users still need to resort to general *JS* instructions (see Figure 5.4).

Finally, ***IE Accelerator*** illustrates "opportunistic augmentation". This functionality of *Internet Explorer (IE)* permits to augment web pages with the *HTML* output obtained through a service request. In the authors' own words: "simply highlight text from any webpage, and then click on the blue *Accelerator* icon that appears above your selection to obtain driving directions, translate and define words, email content to others, search with ease, and more" [Mica]. The resulting *accelerator* can next be shared through the *IE Add-on Gallery* webpage. *Accelerator* is an attractive tool for "opportunistic augmentation", i.e. you are browsing, look at a city and want to see what the weather like in this city is. You have not planned to do so. *Accelerator* permits to highlight the name of the city and feed this data to a weather forecast service (should this be available). Ascertaining the weather at other places would require the same process. From a consumer perspective, *accelerators* are *XML* files.

5.3.2 Our Contribution

Client-Side Open Personalization (COP) provides a model and its instantiation that allows end-users to extend websites' content (a.k.a. *Web Augmentation*). *Web Augmentation* domain is analysed and modelled as a *Domain Specific Language (DSL)*. This model is instantiated as an internal *DSL* built on *JavaScript* in *Sticklet*, a plugin for *Firefox*. During the design of the solution, we have taken into account the insights proposed in the theory of *"Attention Investment"* which have in mind the attention available and required to perform a task.

call by *"find(/ISBN-10: (\d{10})/)"*.

Next sections introduce *Sticklet* by gradually addressing concerns for different audiences: first, conditioned producers, next conditioned consumers, and finally, discretional prosumers. But first, we provide a brief on *JavaScript* to highlight the main programming difficulties to be hidden from end users.

5.4 Web Augmentation: Caring for Producers

This section focuses on conditioned producers, i.e. motivated end-users whose scripts can be potentially consumed by others. The target profile is that of hobby programmers with no knowledge of *JavaScript*. The producer should know about *URLs*, *URL* parameters, and a bit of *HTML* is recommended though not strictly necessary. They do not need to know either *JavaScript* or any other programming language.

The trade-offs exist between expressiveness, freedom, and being general-purpose on the one hand, and usability, learnability, control, and being domain-specific on the other [KPW06]. The challenge is to abstract away from *JavaScript* by identifying *recurrent* abstractions in augmentation scripts. We next restrict the full expressiveness of *JavaScript* to a set of patterns for augmentation which is finally captured through a *DSL*. This section is then about maximizing **expressiveness** without compromising reliability and learnability.

Expressiveness requirements are first captured through domain analysis, and next, framed by the target audience (e.g. conditioned prosumers) [MHS05]. A main output of domain analysis is the feature diagram [KCH+90]. A feature diagram represents a hierarchical decomposition of the main concepts (i.e. features) found in the domain. The diagram also captures whether features are mandatory, alternative, or optional. Figure 5.5 depicts the feature diagram for the domain *"Web Augmentation"*. The diagram states that "a *Web Augmentation* script" includes a *scope* (that sets the ambit of the augmentation), *data extractors* (for variable assignment), an *augmentation lever* (that triggers

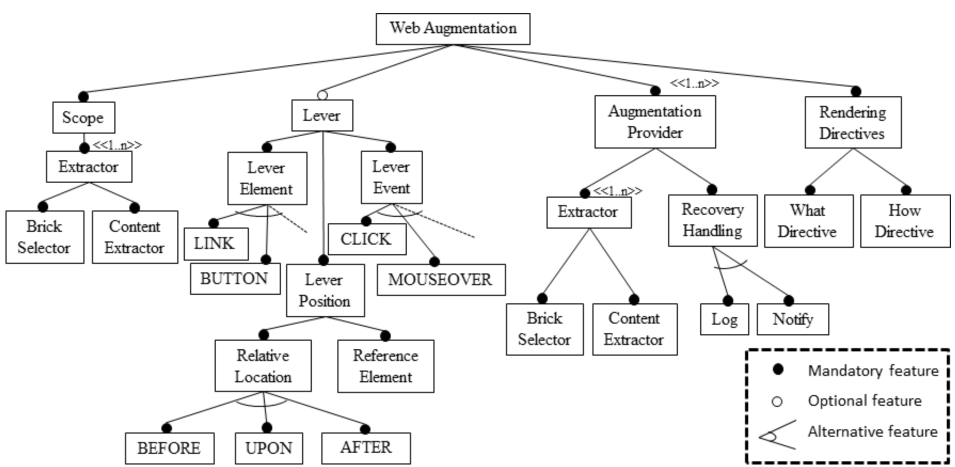

Figure 5.5: Feature diagram for the *Web Augmentation* domain.

the augmentation as such), the *enactment of URL requests*, and the *insertion of its output* in the augmented page. These are the "recurrent abstractions" to be potentially captured as primitives in *Sticklet*.

These features do not come out of the blue but as repeating concerns that are coded time and again. For comprehension purposes, it is convenient to go back to Figure 5.2, and to identify these abstractions on the raw code of *BookBurro*. This permits to better appreciate the abstraction effort (bold stands for features in Figure 5.5):

- **Scope** (line 3). This sets the context where the augmentation occurs: hosting sites that hold certain data. Hosting sites are captured by the *@include* metadata that keeps a URL pattern (line 3). The content of the site is verified through the extractors.

- **Extractor** (lines 9-19). The extractor locates the content to be obtained from the page. This predicate can be a **brick selector** (e.g. the existence of the string *"ISBN-10"* in a certain position in the *DOM* tree, lines 9-10) and/or **a content extractor** (e.g. a string following a certain pattern, line 18).

- **Lever** (lines 21-26). Matching the scope might not directly trigger the augmentation. Augmentation enactment might first require the

user to undertake some actions (e.g. clicking a button, passing the mouse over a certain page region, etc.). These actions are realized as *DOM* events on an *HTML* object: the augmentation lever. An augmentation lever is characterised through three elements: a **lever event,** which is raised by the user on interacting upon a **lever element** which is in turn, placed at a given **lever position**. For *BookBurro*, the lever is *(onClick, link, after the ISBN).*

- **Augmentation Provider** (lines 28-39). *Greasemonkey*'s *API* function *GM_xmlhttpRequest* allows user scripts to get and post data to any site. These data can be retrieved from other web applications (hence, returning *HTML* documents) as well as from web services (which output *XML* or *JSON*). This is one of the main enablers of *Web Augmentation*. But also a main headache. Service fulfilment involves parameter construction, service enactment and error handling. **Recovery handling** for communication pitfalls might need to be considered.

- **Rendering Directives** (lines 34-36). The outcome of service enactment is mashed up into the hosting site. Three output formats are considered: *HTML*, *XML* and *JSON*. This implies both to **select** the desired data from the service outcome as well as to provide **presentation** directives for these data.

Figure 5.5 outlines the main abstractions. Next, these abstractions are realized in a language by looking into variabilities and commonalities in the feature diagram. Variable parts must be specified directly in or be derivable from *DSL* expressions. In the first case, the variants become *DSL* constructs. On the other hand, some alternatives can be hardwired into the *DSL* engine as heuristics. Being heuristics, they might fail and hence, they are not as reliable as if provided by the user. The upside is that they simplify the user's life, hence, improving learnability. For *Sticklet*, we decided *outcomeRendering* and *recoveryHandling* to be hardwired into the interpreter (rationales later). The rest of features are set by the user through

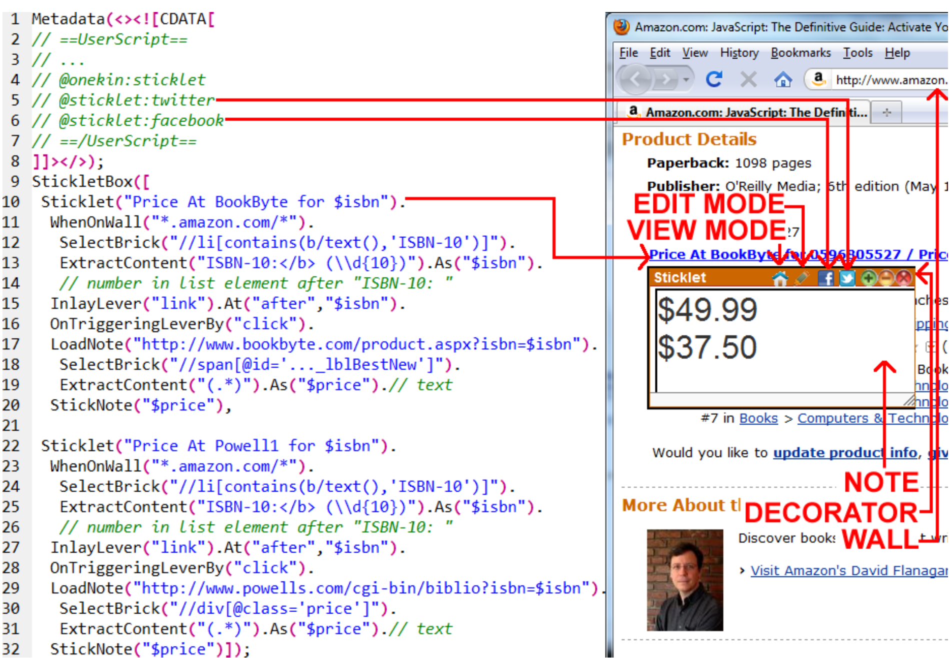

Figure 5.6: *BookBurro* as a *sticklet*.

the DSL. We then need to come up with a set of constructs to specify the rest of the augmentation features.

At this stage, it is most important to devise a metaphor that facilitates the understanding of the *DSL* constructs. Hence, we regard the web as a *wall* to be decorated with *stickers*. Stickers are *notes* (i.e. *HTML* fragments) dynamically obtained from other websites. *Web Augmentation* is then rephrased as fixing *sticky notes* into *walls*. The pair *(wall, sticky note)* conforms an augmentation unit: a ***sticklet***. Hereafter, the term *"Sticklet"* will denote the engine whereas *"sticklet"* will refer to a *Sticklet* expression or augmentation script.

As an example, Figure 5.6 provides the *sticklet* counterpart of the *BookBurro* script (italics denote *DSL* constructs): *WhenOnWall* of *Amazon*, *SelectBrick* and *ExtractContent As* the *isbn* variable, then, *InlayLever At* a given *brick* and, *OnTriggeringLeverBy* a certain event, *LoadNote* from *BookByte*, *SelectBrick price* and, finally, *StickNote*. Figure 5.7 provides the abstract syntax.

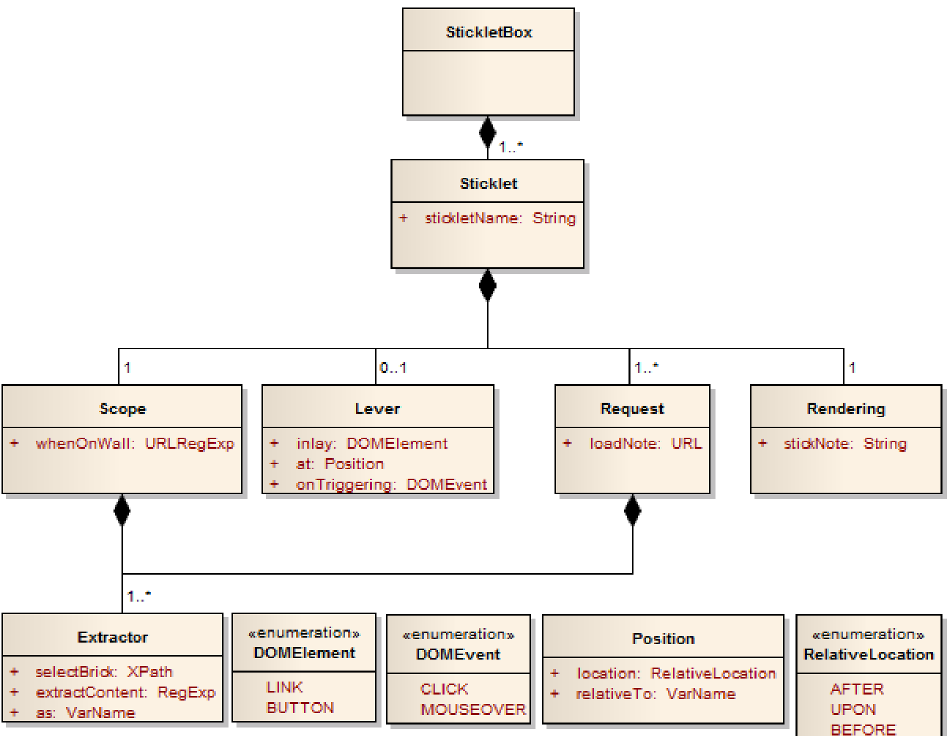

Figure 5.7: *Sticklet*: abstract syntax.

A *StickletBox* comprises a set of *sticklets*. A *sticklet* includes the scope (*WhenOnWall*), the extractors (*SelectBrick*, *ExtractContent*, *As*), the augmentation lever (*InlayLever*, *At*, *OnTriggeringLeverBy*), the augmentation requests (*LoadNote*), and the rendering of the augmentation (*StickNote*). Finally, this abstract syntax is realized through a concrete syntax, either graphically or textually. We opted for a textual *DSL* (see Subsection 5.6.1 for the rationales). The *Sticklet BNF grammar* can be found in the Appendix. Next, we introduce *Sticklet* syntax through examples.

To see the *BookBurro sticklet* at work just type *http://tinyurl.com/cxw9ocy*, and you will be guided through the installation process. Otherwise, follow these steps **(order matters)**:

1. install the *Sticklet* validator:

https://addons.mozilla.org/addon/sticklet/.

2. install the *Greasemonkey* weaver:
   https://addons.mozilla.org/addon/greasemonkey/.

3. install *Sticklet* engine:
   http://dl.dropbox.com/u/6584559/stickletScrambled.user.js.

4. install the *Sticklet* scripts by drag&drop the *scriptName.user.js* file into the browser.

5. edit *Sticklet* scripts using any textual editor (e.g. *Notepad*), the *Greasemonkey* editing facilities, or the inline editor (see later).

5.4.1 Sticklets

We begin with *BookBurro* but now specified as a *sticklet*. This serves to compare the savings in lines of code (and hence, to appreciate the gains in both readability and productivity) between *JS* and *Sticklet*. This *sticklet* is shown in Figure 5.6. The constructs of the *DSL* include: *walls*, *bricks*, *notes* and *levers*.

Walls (line 11). A *wall* comprises those websites whose *URLs* match a given regular expression (*WhenOnWall* clause). They can be regarded as "views" upon the existing websphere. The scope of the *sticklet* is defined by its *wall* as well as by the existence of some *bricks*. For our sample problem, the *wall* expands along those *Amazon* pages that hold an *ISBN brick.*

Bricks (lines 12-14). They are named nodes upon *HTML* documents which are worth singularizing for either data extraction, scoping or layering purposes. A *brick* holds (1) an *XPath* to pinpoint the node (*SelectBrick* clause), (2) a regular expression to extract the node's content (*ExtractContent* clause), and (3), the *brick*'s name (*As* clause).

Notes (lines 17-20). They are expressions that combine text and *bricks* (*StickNote* clause). *Bricks* can be obtained from the *wall* as well as from

URL-addressable web applications (*LoadNote* clause, line 17). For the sample problem, a request is made to *BookByte* where *URL* parameters are obtained from previously extracted *bricks* (e.g. *$isbn*). The outcome is used to pinpoint a new *brick: $price.* Finally, *bricks* from different sources are used to conform the *note* (*StickNote* clause). *Notes* are framed by a decorator. *Notes* can be dragged around, expanded to fit, minimized, or just closed. *Notes* might be readily stuck as soon as the user enters the *wall*. However, this is not always the desired behaviour. You might be looking at *Amazon* with no intention of buying a book. Readily sticking *notes* could lead to cluttered pages, being contra-productive and inefficient by forcing the enactment of the *sticklet*'s *note* with no purpose. Therefore, fixing a *note* might involve an additional user intervention: acting on a *lever*.

Levers (line 15-16). They permit to obtain *notes* on demand. *Levers* are named after the *sticklet* name (e.g. *"Price at BookByte for $isbn"*) where variables (e.g. *$isbn*) are resolved at runtime. *Levers* are positioned according to *bricks*. For *BookBurro*, a *lever* (realized as a *link*) is inlayed *after* the brick *$isbn*. Other options include *"before"* and *"upon"*, where the latter replaces the *brick* by the *lever*. On acting upon the *lever*, a *URL*-addressable web application is enacted. For *BookBurro*, the *lever*'s event is a *click*, though any *DOM* event is permitted. On clicking, the *BookByte* request is conducted; next, the book price is obtained and finally, the *note* is rendered.

Two important remarks about *bricks*. First, *bricks* identify entities of interest. In the sample case, the *$isbn brick* is atomic. But entities might not be either atomic (the associated *XPath* expression returns an *HTML* fragment rather than an atomic value) or unique (the expression outputs a list of nodes). The *HTML* fragment *CustomerReview* at *Amazon* provides an example. First, an *Amazon* page might hold distinct *CustomerReview*. Second, a review is a compound, i.e. it holds a score, a headline, a description, a reviewer and a review date. That is, each instantiation of the *$customerReview* would hold a different, structure-rich *HTML* fragment.

> ***Bricks* hold *HTML* fragments. This permits to use *XPath* to obtain *bricks* out of these compound *bricks*.**

An example follows:

```
SelectBrick("//tr...").
     ExtractContent("(.*)").As("$customerReview").
SelectBrick("$customerReview//span...").
     ExtractContent("\\d").As("$score").
SelectBrick("$customerReview//div...").
     ExtractContent("by (.*)$").As("$reviewer")
```

The second remark is about the three-fold role played by *bricks:*

> ***Bricks* can serve to (1) extract data from pages, (2) pinpoint locations for *lever* positioning, and (3), determine the number of *sticklet* instances.**

The latter requires further explanations. Operationally, *sticklets* can be regarded as triggers (e.g. on loading an *Amazon* page with an *isbn*, inlay a *lever*). The implicit event is not "on loading an *Amazon* page" but "on obtaining an *ISBN* from an *Amazon* page". This is a paramount difference from the perspective of the operational semantics of *Sticklet* (see Subsection 5.4.7). It implies that if the *Amazon* page does not contain an *ISBN*, the *sticklet* does not apply. Likewise, if the *Amazon* page contains several *ISBN*s then, distinct *sticklet* instances will be fired (hence, placing different *notes* by each *ISBN*). Notice that if *$customerReview* is introduced in addition to *$isbn*, a *sticklet* instance will be triggered for each combination of $[\$customerReview\, x\, \$isbn]$. For our sample case, this means ten instantiations since the page for the sample book holds an *isbn* (i.e. a single node satisfies the *XPath* associated with *$isbn*) and ten reviews. This in turn, implies that ten *notes* would have been generated and placed by the *lever*. More to the point, *$customerReview* need to be introduced even if you are interested only in part of its information (e.g. the

score and the reviewer). Explicitly naming this entity indicates that *score* and *reviewer* are not two independent notions but they belong to a higher concept (i.e. *customerReview)* in which terms the *sticklet* is described*:* the number of *customerReview* (and not *scores* or *reviewers*) determines the number of times this *sticklet* is to be triggered (more in Subsection 5.4.7).

5.4.2 StickletBox

A *sticklet* accounts for a pair *(a wall, a sticky note)*. However, a single *sticklet* might not be enough. For instance, the *BookBurro* script might be conceived as enhancing not just *Amazon* but a set of online bookshops (e.g. *Amazon, BookByte, Powell*). The price-comparison *note* is available for any of these bookshop*s*. Since a *sticklet* supports a pair *(a wall, a sticky note)*, the *BookBurro* functionality requires six *sticklets*: *(Amazon, BookByte's price)*, *(Amazon, Powell's price)*, *(BookByte, Amazon's price), (BookByte, Powell's price)* and the like. This grounds the notion of *stickletBox*:

> **A *stickletBox* is a set of *sticklets* that stand for a meaningful unit of augmentation.**

Therefore, a *stickletBox* permits the very same wall (e.g. *Amazon*) to receive *notes* from different websites (e.g. prices at both *BookByte* and *Powell*). This begs the question of whether those *notes* should be obtained simultaneously (and hence, displayed in the same *note*) or not. This is regulated by *levers*. *Levers* are characterized through *bricks* (e.g. *$isbn*).

> ***Bricks* with *both* the same name and associated *XPath* denote the very same position in the *wall*. Therefore, *notes* from different *sticklets* but attached to namesake *bricks* are simultaneously obtained and rendered.**

BookBurro provides an example (see Figure 5.6, lines 15 and 27). Two *sticklets*, *"Price At BookByte for $isbn"* and *"Price At Powell1 for $isbn"*,

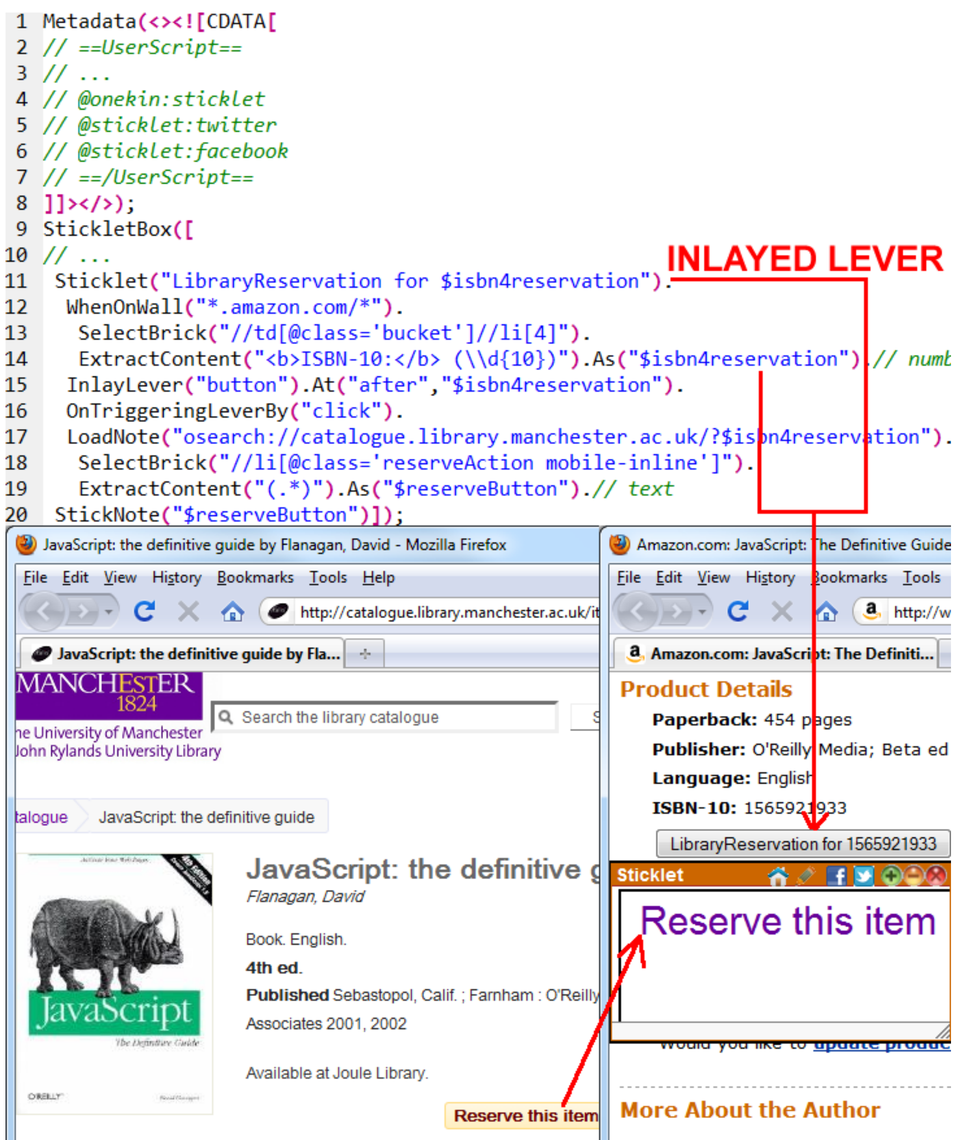

Figure 5.8: *Sticklet* for augmenting *Amazon* with the book reservation at the Manchester University Library.

introduce a *lever* which is associated with the *$isbn*. Therefore, their outputs are simultaneously rendered in the very same *note*. By contrast:

> **If two *notes* account for different information needs then, two differentiated *levers* permit to resolve these needs separately.**

"LibraryReservation for $isbn4reservation" illustrates this case (Figure 5.8). We want to know whether the book at *Amazon* is available at the *Library of Manchester University*, and reserve it[4]. The augmentation captures the *"reserve"* button at this guest website, and sticks it on the *Amazon* page. Since this need is not geared towards purchasing the book, a separate *lever* (line 15) is inlayed by the *$isbn4reservation brick*.

Finally, note that *sticklets* can be added/removed at any moment. *Sticklets* are self-contained, no coupling exists among *sticklets* kept on the same *stickletBox*. This accounts for maintainability and provisionability (see later).

5.4.3 The Issue of Entity Linkage

Sticklets contextualize data from different websites into a single workspace: the *wall*. Contextualization implies the existence of a "sharing notion" between the wall (e.g. *Amazon*) and the note providers (e.g. *BookByte*). In the previous example, this notion was the *ISBN:* extracted from *Amazon* as a *brick*, and communicated to *BookByte* as a *URL* parameter. However, this is not always so easy. An entity might exhibit distinct representations. A book can be denoted by the *ISBN*, the pair (title, author), an *ad-hoc* code, or even the book's cover can be the only reference to a book. As an example, consider the *www.walmart.com* website. *Walmart* also sells books. However, *Walmart*'s URLs are not based on *ISBNs* but on an internal code. For instance, the *URL* for the book used

[4] You can find this service for our book example at *http://catalogue.library.manchester.ac.uk/items/2049288.*

as an example in *Amazon* is *http://www.walmart.com/ip/13443765,* where the ending number has nothing to do with this book's *ISBN*. This raises a mismatch between how the entity is represented in the *wall* (e.g. *Amazon*) and how the entity is captured in the *URL* parameter (e.g. *Walmart*).

This problem also arises in databases (known as *Record Linkage*) when two data sets need to be joined and they do not have a unique database key in common (e.g. passport number vs. national insurance number) [Win06]. In the *Semantic Web*, where resources are described through *URLs*, they also encounter the so-called *Coreference problem*, i.e. ascertaining where two distinct *URL*'s stand for the same entity [JGM07]. The use of mediate ontologies and shared resources such as the *DBpedia* [BLK$^+$09] can help to provide a common ground to facilitate integration. In ontology mapping, the challenge is to discover automatically alignments between entities described in different ontologies, exploiting lexical similarities, lattice structure or instance classification learning techniques [SBVG10]. These approaches tend to be time consuming and, in some cases, imply user intervention. However, we strive to minimize both user intervention and elapsing times (remember, shortage of attention). As a result, *Sticklet* does not provide any module for "entity linkage" but resorts to mapping and search.

Linkage through searching. Since *ISBN*-linkage does not work, we will mimic what users would do: go to *Walmart* and conduct a local search for e.g. the book's title. This implies to access programmatically the search facilities of the *Walmart* website. Fortunately, the *OpenSearch* Discovery initiative has already standardized this process [AA05]. An *OpenSearch* description document can be used to describe the web interface of a search engine. This description holds parameterized *URL* templates that indicate how the search client should make search requests (the *<Url>* element). This document is referenced in the search page of the website through a link (look at the source code of *http://www.walmart.com/*)[5]:

[5]Making explicit how to query a website permits to introduce "custom search engines" in browsers [eTe10]. When viewing an *HTML* page that includes the *<link>* tag above,

```
1  Metadata(<><![CDATA[
2  // ==UserScript==
3  // ...
4  // @onekin:sticklet
5  // @sticklet:twitter
6  // @sticklet:facebook
7  // ==/UserScript==
8  ]]></>);
9  StickletBox([
10 // ...
11  Sticklet("Walmart").
12   WhenOnWall("*.amazon.com/*").
13    SelectBrick("//div[@id='divsinglecolumnminwidth']").
14    ExtractContent("(.*)").As("$book").// text
15    SelectBrick("$book//span[@id='btAsinTitle']/text()").
16    ExtractContent("(.*)").As("$title").// text
17    SelectBrick("$book//td[@class='bucket']//li[4]").
18    ExtractContent("<b>ISBN-10:</b> (\\d{10})").As("$isbn").//
19   InlayLever("link").At("after","$title").
20   OnTriggeringLeverBy("click").
21   LoadNote("osearch://www.walmart.com/?$isbn").
22    SelectBrick("//div[@class='PricingInfo clearfix']/div[1]").
23    ExtractContent("(.*)").As("$price").// text
24   StickNote("$price")]);
```

Figure 5.9: Entity linkage through searching: *Walmart* is queried about rather than requested for a specific *URL*.

```
<link rel="search"
type="application/opensearchdescription+xml"
title="Walmart.com"
href="http://www.walmart.com/walmartdotcom.xml"
/>
```

This technology is used in *Sticklet* to resolve entity linkage by extending the semantics of the clause *LoadNote* . So far, *LoadNote* keeps

browsers can either automatically collect the *OpenSearch* file (e.g. *Chrome*) or highlight some icon of the browser search box (e.g. *Firefox*) that permits users to explicitly add the current site as a source for their queries. You can check this out by navigating to *www.walmart.com*. On detecting `<link rel="search">` in the page source, *Firefox* faintly highlights the little arrow of its search bar. Click on this arrow, and observe how a new menu item prompts to include *Walmart* as a new custom search engine.

a *URL* expression which is parameterized by *bricks* extracted from the *wall*. Now, *LoadNote* can also be instructed to transparently conduct an open search by turning the *"http"* protocol into the *"osearch protocol"* (see Figure 5.9, line 21):

> ***LoadNote*** **can be commanded to search into the note-provider website to find the wall-notion counterpart.**

This process is conceived as a kind of protocol in the sense that a conversation is initiated between the *Sticklet* agent and the guest website (e.g. *Walmart*). The "osearch protocol" commands *Sticklet* to go to *www.walmart.com*, locate the *OpenSearch* document as a *<link>* in the source page, recover the *<Url>* element, construct the search request using the associated template (as an attribute of the *<Url>* attribute), process the output (an *HTML* page), and finally, come up with the *URI* of the sought resource. Shouldn't the website contain an *OpenSearch* document then, *Sticklet* goes back to the *http* protocol: loads the *URL* (without parameters), attempts to locate a search box in the returned page, and finally, feeds the search box with the search parameter (e.g. *$title*). A final hardwired strategy is to conduct the search in *Google* (i.e. *"site:www.walmart.com AND $isbn"*) and retrieve the first link.

No matter the way, *Sticklet* retrieves a page. This page can stand for the sought resource (and then, the process ends) or deliver a list of resources that match the query parameters. This is the case when the *osearch* protocol is used for *Walmart* for the title *"JavaScript: The Definitive Guide"*. Figure 5.10 shows the output. In this case, *Sticklet* applies a set of heuristics to ascertain the *URI* of the sought resource. Since *LoadNote* has to return a single *URI*, the algorithm focuses on those *DOM* nodes that hold a *URI*. Next, the algorithm identifies those text nodes containing text values used in the query on the assumption that the values used in the query will typically appear with higher probability in the list of results than in other lists of the page. Once the list of books is singularised from other lists in the *Walmart* page, it rests to identify the sough resource within

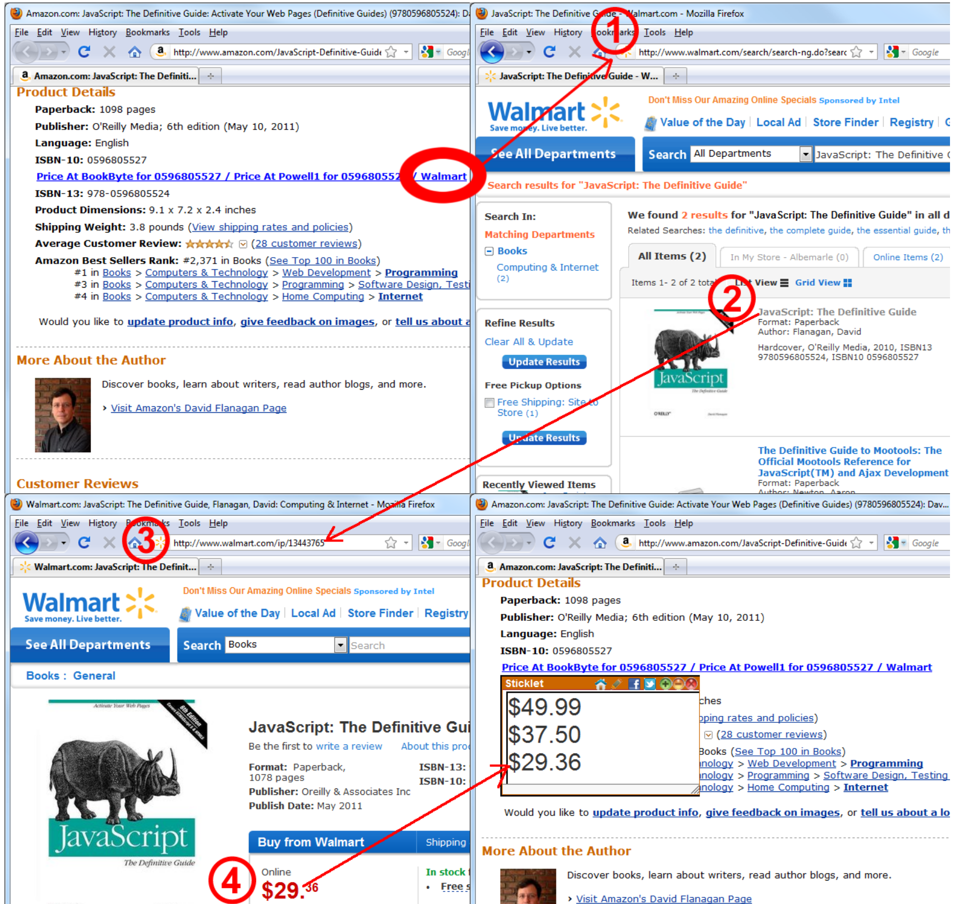

Figure 5.10: Entity linkage through searching. Clicking the *lever* makes *Sticklet* initiate a conversation with *Walmart*: (1) *Walmart* is queried about the title "*JavaScript: The Definitive Guide*" using *OpenSearch; (2) Sticklet* looks for the URL of interest among the URLs held in the returned page; (3) *Walmart* is requested for this URL; (4) *Sticklet* digs the price out of this second page. Notice that returned pages are just for *Sticklet* consumption, kept transparent from the user.

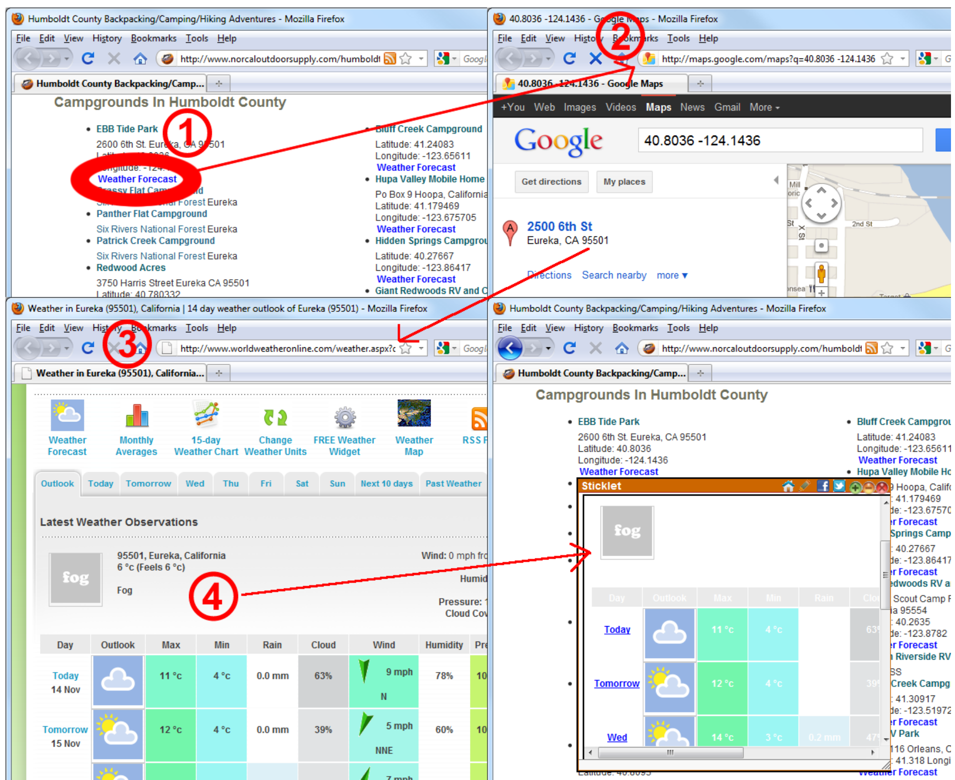

Figure 5.11: Entity linkage through mapping: (1) clicking the *lever*, (2) causes a *GoogleMaps* to be loaded which contains the *$zip*; (3) this zip is used to build a request to *worldweatheronline* which holds the *$weather* note; finally (4), this *HTML* fragment is sticked. Only the final step is visible to the user.

this list. If the *URI* nodes at *Walmart* are accompanied by additional information (sibling nodes), then *Sticklet* will compare those sibling nodes as a unit with the compound *brick* that holds the query parameter (e.g. *$title* and *$book*, respectively), should such *brick* be available. If this process does not filter a single *URI* then, *Sticklet* will return the *URI* in the first position. This process is built into *Sticklet* as the semantics of the *"osearch"* construct.

It could be argued why not to conduct the query in terms of *$book* in the first place rather than in terms of a book property such as *$title*. The reason is that *$book* might contain a broad range of data other than

```
1  Metadata(<><![CDATA[
2  // ==UserScript==
3  // ...
4  // @onekin:sticklet
5  // @sticklet:facebook
6  // @sticklet:twitter
7  // ==/UserScript==
8  ]]></>);
9  StickletBox([
10  Sticklet("Weather Forecast").
11   WhenOnWall("www.norcaloutdoorsupply.com/*").
12    SelectBrick("//div[@class='panel']//li").ExtractContent("Latitude").As("$campground").
13    //  in list element
14    SelectBrick("$campground").ExtractContent("Latitude: (-?\\d{1,3}\\.\\d{3,6})").As("$lat").
15    // text and/or number after "Latitude: "
16    SelectBrick("$campground").ExtractContent("Longitude: (-?\\d{1,3}\\.\\d{3,6})").As("$lon").
17    // text and/or number after "Longitude: "
18   InlayLever("link").At("after","$camping").
19   OnTriggeringLeverBy("click").
20   LoadNote("http://maps.google.com/maps?q=$lat $lon").
21    SelectBrick("//span[@class='...']/span").ExtractContent("(\\d+)").As("$zip").// number
22   LoadNote("http://www.worldweatheronline.com/weather.aspx?q=$zip").
23    SelectBrick("//div[@class='...']").ExtractContent("(.*)").As("$weather").// text
24   StickNote("$weather")]);
```

Figure 5.12: Entity linkage through mapping. *GoogleMaps* acts as a mapper from geo coordinates to zip codes.

that used by the user in the query. In the example, the user characterizes books in terms of title and isbn, thought book nodes (as pinpointed by the associated *XPath* expression) encompasses a wider range of other data (e.g. authors). Querying by *$book* (better said, the content of *$book*) might be too stringent, causing the query to output no result at all. Therefore, users decide the properties to search for, and next, *Sticklet* applies the heuristics above to filter out the sought *URI*.

Linkage through mapping. This approach resorts to intermediate websites to act as mediators.

> ***Notes* from distinct websites can be loaded that help to map the wall notion into its counterpart into the note-provider website.**

Figure 5.11 provides an example of a website about hiking at *North California*. Camping spots are identified by their geo coordinates. We would like to augment this site with the weather forecast as available at *www.worldweatheronline.com*. This site can be queried using different criteria (e.g. the zip code, the city name) but not through geo coordinates.

Geo coordinates is not a property you can search for. Fortunately, *GoogleMaps* can act as a mediator. Figure 5.12 provides the *sticklet* code. Mediators are introduced as note providers (lines 20-21). Worth noticing: (1) the multiple instantiation of the *$campground brick* at the hosting site, and (2), the use of two *bricks*, *$lat* and *$lon*, to query *GoogleMaps*.

5.4.4 The Issue of XPath Complexity

XPath is outside the competences of our target audience. Assistance is required that hides this complexity from *producers in writing their sticklets* as well as from *consumers in understanding third-party sticklets*. The question is how to specify an extraction pattern without knowledge of *XPath* or regular expressions, or understanding *HTML*. Answers include the use of:

- heuristics, that permit to refer to buttons, links, and other web page elements in terms of nearby text (termed "text constraint patterns"). In this way, users do not use *XPath* to pinpoint the desired data but just indicate the nearby text (e.g. the data that is after the "ISBN" text), and let the system guess the right location. This approach is illustrated by *Chickenfoot* [BWR$^+$05] and *CoScripter* [LHML08] (Section 5.3.1 illustrates the *BookBurro* example using *Chickenfoot*). For our purposes, this approach brings two main benefits: learnability (for producers) and understandability (for consumers). The downside is efficiency. Text constraint patterns imply processing the whole document on the search for the patterns *everytime* a new page is loaded. For complex pages/patterns, such process can incur in noticeable delays every time the script is run.

- programming-by-example, where users first highlight elements of the web page which serve to infer the matching pattern (e.g. the *XPath* expression). Systems ***re**form* [TDD$^+$09], *Karma* [TKS11] or *MashMaker* [EBG$^+$07] use programming-by-example. For

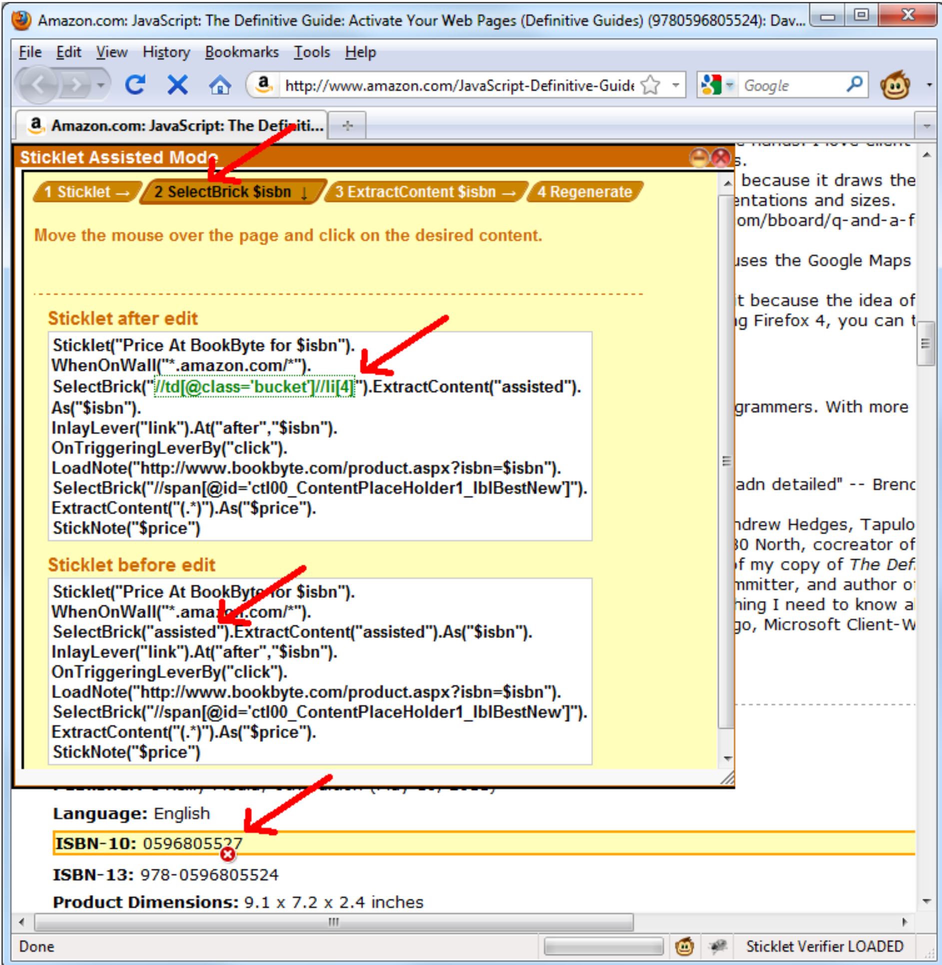

Figure 5.13: The *assisted* mode. The user is guided through the resolution of the unbounded clauses by highlighting the selected *HTML* region in the hosting page.

our purposes, this approach brings efficiency since the inferring algorithm is only run at definition time. This option eases production but still leaves consumers to face *XPath* expressions.

Sticklet explores a hybrid approach:

> ***Sticklet* permits to substitute *XPath* expressions by the value *"assisted"*. This makes *Sticklet* differ till run time the binding of these clauses, and resolve them visually.**

As an example, consider the *BookBurro* script but now using *assisted* as the value for *SelectBrick* and *ExtractContent* clauses. The semantics of *assisted* works as follows (see Figure 5.13):

1. on loading the *Amazon* page, the *sticklet* is enacted. The engine detects that the *sticklet* is not fully resolved, i.e. it contains some assisted-valued clauses. Hence, the engine layers a panel for the assisted editing. The panel includes: a progress tracker, the regenerated *sticklet* (*"Sticklet after edit"*) and the current *sticklet* (*"Sticklet before edit"*). The progress tracker provides an ordered way to resolve each of the unbounded clauses. For the sample problem, this tracker provides four stages, one for each unbounded clause, and finally, the regenerate option.

2. the user clicks on the first unbounded clause (e.g. *SelectBrick $isbn*). This makes *Sticklet* intersperse a grid-like structure on top of the current *DOM* tree. As the user moves the cursor around the screen, the *DOM* node under the current cursor location is highlighted. By clicking, the user feeds the inferring algorithm with the selected node, and *Sticklet* highlights all the nodes that fulfil the extraction pattern generated so far. For the *ISBN*, there is only a single node in the hosting page that fulfils this notion, and hence a single interaction might suffice. However, the augmentation might impact different nodes of the hosting page. The *brick $customerReview* provides an example. This *brick* might be instantiated several times throughout

the page. This requires the user to pinpoint distinct reviews till the correct extraction pattern is abstracted by the inferring algorithm. [6]Heuristics from *XPath* generation are inspired in work described in [PD10, ÁPR+10, LE07].

3. Next, the following node of the progress tracker becomes active (i.e. *ExtractContent*). Now, the outcome is a regular expression. The user selects the content of interest, and *Sticklet* obtains an expression that matches this content. When *bricks* act as filters, the content might be the filtering criteria (e.g. books whose price is 103). That is, the expression is a constant (e.g. "103"). This makes the engine look for nodes whose content is "103". If this is the desired behaviour, just skip the *ExtractContent* step. This makes the content of the current node become the constant value of *ExtractContent.*

4. Finally, regenerate the script, i.e. the script is automatically updated and re-installed.

The *"assisted"* option could suggest that the explicit introduction of *XPath* is no longer needed. However, two scenarios advice to keep this option open. First, the inferring algorithm might fail to extract the correct data[7]. Both *Chickenfoot* and ***re****form* warn about the heuristic nature of this process. Keeping this option open permits at least to ask for assistance to an *XPath* expert. Second, some data might not have a rendering counterpart, hence, no way for the user to pinpoint this data out. A common example is extracting *URLs*. *URLs* tend to be provided as attributes of *HTML* anchors. The anchor's content is what you see while

[6]*Sticklet* abstracts away from absolute paths into a relative path that strives to capture the essence of the rendering of *$customerReview* (e.g. *"//tr[2]/td[3]/div/a"*). *Sticklet* highlights in the canvas all the nodes that account for the so-obtained *XPath*. Should some node be missing, the user can click on the missing node and let *Sticklet* regenerate the *XPath* expression. Once all nodes of interest are highlighted, the user clicks in the next step of the progress tracker, and the so-generated *XPath* becomes the value of the *SelectBrick* clause as visualized in the *"Sticklet after edit"* panel (see Figure 5.13).

[7]So far, the algorithm can learn from positive but not negative examples as suggested in [TDD+09].

the anchor's *URL* is what you might want. In this case, *"assisted"* is of little help since all you can highlight is the anchor's content. Therefore, *Sticklet* keeps open the possibility of explicitly providing the *XPath* expression.

So far, we address the complexity of *XPath* from the producer perspective. The "assisted" approach hides this complexity for producers but consumers are still exposed to *XPath* expressions. Understandability is a major benefit of text constraint patterns (e.g. the data that is after the "ISBN" text) that we lost when moved to *XPath*. In an attempt to overcome this limitation,

> ***Sticklet* automatically generates a comment by each *XPath* expression that re-phrases in natural language-like terms the types of *XPath,* and the regular expression pattern.**

Figure 5.6 provides an example (lines 14 and 26). In this way, *Sticklet* strives to bring the best of both worlds. Using programming-by-example, *Sticklet* obtains efficient and accurate *XPath* expressions without imposing a major burden on producers. Generating pattern-like comments, *Sticklet* attempts to be as understandable as the alternative of directly providing text constraint patterns.

A final comment on maintainability. Unlike text constraint patterns, *XPath* expressions are fragile: changes on the structure of the underlying Web page can make the *XPath* expression stop working. If *Amazon* changes the *ISBN* location, the *Sticklet* expression will fail to recover the proper data. This is true. But the overhead of re-generating the script is affordable: edit the broken *sticklet*, substitute the *XPath* expression by "assisted", and finally, regenerate the *xpath*. In our opinion, this maintainability burden compensates for avoiding working out the location of data everytime the script is run as in text constraint patterns.

5.4.5 The Issue of Non-HTML Sources

Notes can be obtained from sources other than *HTML* pages. *RSS* feeds or *URL*-addressable programmatic interfaces deliver data-centric *XML*

```
1  Metadata(<><![CDATA[
2  // ==UserScript==
3  // ...
4  // @onekin:sticklet
5  // @sticklet:facebook
6  // @sticklet:twitter
7  // ==/UserScript==
8  ]]></>);
9  StickletBox([
10 // ...
11  Sticklet("Comments on $title").
12   WhenOnWall("*.amazon.com/*").
13    SelectBrick("//span[@id='btAsinTitle']/text()").
14    ExtractContent("(.*)").As("$title").// text
15    SelectBrick("//div[@class='content']//td[1]/table/tbody/tr/td[1]/div").
16    ExtractContent("(.*)").As("$AmazonAverageReview").// text
17    SelectBrick("//li[contains(b/text(),'ISBN-10')]").
18    ExtractContent("ISBN-10:</b> (\\d{10})").As("$isbn").// number in list element after "ISBN-10: "
19   InlayLever("link").At("after","$isbn").
20   OnTriggeringLeverBy("click").
21   LoadNote("http://www.goodreads.com/search?q=$isbn").
22    SelectBrick("//div[@class='review']").
23    ExtractContent("(.*)").As("$GoodreadsReview").// text
24   StickNote("<h1 style='text-align:center;'>Comments for the book $title</h1><br><br>"+
25             "<h2>Average Customer Reviews</h2>"+
26             "<div style='overflow:hidden;'>$AmazonAverageReview</div><br><br>"+
27             "<h2>Reviews</h2><ul><li>$GoodreadsReview</li></ul>")]);
```


Figure 5.14: *Note* rendering. Default rendering can be supplemented with *HTML* directives. The figure depicts (a) the default rendering (i.e. no *HTML* tags) and (b), the *HTML*-enriched outcome as specified in the *sticklet* (lines 24-27). The user-provided *HTML* directives prevents the note *$AmazonAverageReview* from overlapping with the *$GoodreadsReview* note.

or even, *JSON* [Cro06]. Being an agent, *Sticklet* can consume these documents. For instance, the previous *NorthCalifornia* camping example could have resorted to an *XML*-based service to obtain the ZIP out of the geo coordinates. Lines 20-21 in Figure 5.12 might be replaced by:

```
LoadNote(
"http://maps.google.com/maps/geo?output=xml&q=$lat,$lon").
SelectBrick("//PostalCodeNumber").
ExtractContent("(\\d+)").As("$zip")
```

The problem is that *XML/JSON* documents are thought for agent consumption rather than human consumption. Human consumption implies reading and interacting via *HTML* documents. Specifically, *Sticklet* expects user interaction to select *bricks* and render *notes*. If the source of the *note* happens to be an *XML/JSON* document, *Sticklet* needs first to convert this document into *HTML*.

> **For *XML/JSON* sources, *Sticklet* applies basic rendering templates to the returned *XML/JSON* document.**

In this way, users select/read *bricks* in the very same way regardless of whether the original representation was *HTML*, *XML* or *JSON*.

5.4.6 The Issue of Note Rendering

StickNote commands the rendering of a *note*. A *note* is an expression that combines text and *bricks* (i.e. *HTML* fragments) from potentially different websites. Each website has its own rendering (i.e. set of *CSS*). The question is how this bulk of disparate *HTML* fragments can deliver "a harmonious note". This endeavour requires *HTML* skills. We decided to remove such burden from the user, and hardwire some rendering heuristics into *Sticklet*.

> ***Sticklet* holds a set of heuristics that provide "good-enough rendering" for sticky notes. In addition, users can supplement the *note* with *HTML*-sanitized markup.**

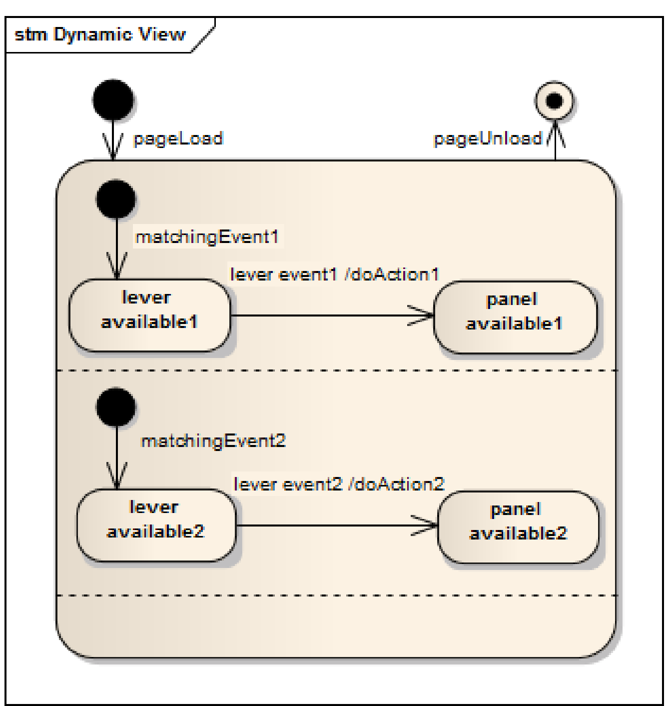

Figure 5.15: The operational semantics of *Sticklet*. *AND* states are separated by a dotted line.

Let's augment *Amazon* with reviews from *www.goodreads.com*[8] (see Figure 5.14). Specifically, the *note* gathers the following *bricks* (lines 13-18 and 22-23): the book title (*$title*), the average customer reviews from *Amazon* (*$AmazonAverageReview*) and reviews from *GoodReads* (*$GoodreadsReview*). These *bricks* can be provided in quick succession without *HTML* ornaments. In this case, *Sticklet* will decide the best rendering (see Figure 5.14 (a)). Heuristics are inspired in those applicable for the automatic rendering of pages through small-screen devices such as mobile phones [BKGM^{+}02, XLH^{+}09]. As a general norm, *bricks* are rendered along the *CSS* directives of their source websites (in this example, those of *Amazon* and *GoodReads).*

[8]Comments for our sample book can be found at *http://www.goodreads.com/book/show/1617424.JavaScript.*

We are aware these heuristics fall short to account for sophisticated results. At worst, the rendering looks like a simple list with no additional ornament. But the main argument here is that *Sticklet* is designed for *HTML* ignorants. Amateurs love to do things by themselves while accepting good-enough results. Forcing users to provide the rendering by themselves would sacrifice better appearance for self-sufficientness.

All in all, *Sticklet* also permits to introduce *HTML* directives. Similar to the editing of wiki articles (and on similar grounds), *notes* can contain basic *HTML* tags. Due to security reasons, *HTML* is sanitized. This basically means that *JavaScript* is not permitted (restrictions are similar to those of *Wikipedia*[9]). Two remarks about *bricks*. First, *bricks* are *HTML* fragments whose *CSS* classes are inherited from their source websites. Those classes can be overridden by *HTML* directives explicitly given in the *note*. Second, *bricks* which can potentially be instantiated several times (e.g. *$GoodreadsReview*) are regarded as lists, and hence, they should be enclosed using a directive for lists (e.g. *<ul>*, see line 27). Otherwise, all instances are rendered as a single row.

5.4.7 The Operational Semantics of Sticklets

This subsection outlines the operational semantics of *sticklets* interpreted as sequences of computational steps. These sequences then *are* the meaning of the *"sticklet"* construct. Augmentation proceeds along two states (see Figure 5.15): the *"lever available"* state and the *"panel available"* state. Transitions proceed as follows:

- on **page loading**, the *URL* of the loaded page is checked against the regular expression of the *WhenOnWall* clause. If met, the engine raises a matching event occurrence for each combination of *brick* instances that fulfil the matching conditions set by the *bricks*. Hence, the event payload is a combination of the *brick* instances. If you have *bricks $a, $b* and *$c* which are matched in the current *wall* 2, 3 and 5

[9] *http://meta.wikimedia.org/wiki/Help:HTML_in_wikitext*

times, respectively, then, the number of matching event occurrences will be 30: *occurrence($a1,$b1,$c1)*, *occurrence ($a2,$b1,$c1)*, and so on. Each matching event occurrence gives rise to a *sticklet* instance (represented as an AND state in Figure 5.15).

- on rising a **matching event**, the engine moves to the *"lever available"* state. This causes the rendering of the *lever* along the behaviour defined in clauses *InlayLever*, *At* and *OnTriggeringLeverBy,*
- on rising the **lever event** (e.g. mouse over, click, etc.), the *LoadNote* service is enacted. The system moves to the *"panel available"* state where the *note* is rendered,
- at any moment, **page unload** causes the current page stop displaying. This ends the augmentation.

Worth noticing is the parallelism behind this semantics. First, matching event occurrences are handled in parallel as denoted by the *AND* states in Figure 5.15. If you have 30 matching occurrences then, 30 transitions to the *"lever available"* state will happen. Second, some of the previous transitions might entail the same *lever* (e.g. *sticklets "Price at BookByte for $isbn"* and *"Price at Poweel1 for $isbn"* involve the same *lever*). In this case, the very same *lever* event is shared among these two *AND* states. Raising this event causes both states to transit to the *"panel available"* substate, and jointly render their outputs as a single *note*. In short,

> **The rule-like semantics of *sticklets* go beyond improving modularity to account for parallelism and atomicity. Efficiency wise, the speed of a *stickletBox* augmentation is that of its slowest *lever*/matching transition.**

The latter entails that if collecting the price from ten online bookshops that share the same *lever*, clicking on this *lever* will cause nine parallel *HTTP* requests (the tenth bookshop is the *wall*). The *note* is constructed gradually

as answers arrive. This means that the order in which prices are displayed in the *note* might vary depending on the traffic load of the sites.

Sticklets as rules might suffer from similar problems as those of triggers in active databases [PD99]:

- termination (is rule processing guaranteed to terminate?). *Sticklets* do always terminate since they cannot raise triggering events (i.e. matching events or *lever* events) that enact other *sticklets*.

- confluence (i.e. is the result of rule processing independent of the order in which simultaneously triggered rules are selected for processing?). Acting on the same *DOM* tree, *sticklets* could potentially suffer for confluence. We refer to this problem in Subsection 2.3.4. Traditional scripts are enacted sequentially based on the order they were installed. This implies changes made by the first script *are visible* to ulterior scripts. Two types of dependencies arise: read dependency (a script can accidentally read data written by a previous script), and write dependency (the injection point can be displaced by the writing of a node made by a previous script). As a result, the very same set of scripts can deliver different outcomes depending on the order they were installed. *Sticklet* addresses this issue (technical details at [DAI10]). Read dependencies are obviated by making *sticklet* changes transparent to other *sticklet*s. *Sticklets* can only access the raw *DOM* (i.e. the *DOM* corresponding to the hosting page) previous to being updated by any *sticklet*. No way for a *sticklet* to see changes conducted by other companion *sticklets*. As for write dependencies, they are avoided by preventing *sticklets* from altering the basic structure of the hosting page. Augmentation can only add new anchors (i.e. *levers*) and *notes*. In this way, *sticklets* do not change the position of the data in the hosting page, hence, avoiding breaking companion *sticklets*.

- observable determinism (i.e. is the effect of rule processing as observed by a user of the system independent of the order in

which triggered rules are selected for processing?). In databases, this notion seeks to extend the notion of confluence beyond the boundaries of the database itself. In Web Augmentation however, the notion of confluence and observable determinism coincides since the shared resource and the observed resource is the same: the *HTML* page.

Similar semantics if handwritten in *JavaScript* scripts would require clumsy algorithms. *Sticklet* handles this control complexity automatically, consistently, and formally.

5.5 Web Augmentation: Caring for Consumers

Previous section focuses on producers. The challenge was to find a balance between expressiveness and learnability/reliability. Now, we introduce consumers as main stakeholders. The consumer profile is that of a computer literate with e.g. basic knowledge about *MS Word* or installing add-ons for *Firefox*. Now, scripts are no longer for self-consumption but for use by a large number of users with varying needs. Being "for use by a large number of users" moves to the forefront trustworthiness. Catering for "varying needs" boosts maintainability. This section presents how *Sticklet* tackles these concerns.

5.5.1 Trustworthiness

Trustworthiness refers to the assurance that a system deserves to be trusted, i.e. that it will perform as expected despite environmental disruptions, hostile attacks, and the design and implementation errors [BHP+06]. This is a must for consumption to become viral. Mechanisms to improve trustworthiness include fault prevention, fault tolerance and fault removal.

Error Type (HTTP code)	**HTTP Error Handling Strategy**
Client-side Recoverable (408, 413)	Retry in a short period
Client-side Unrecoverable (400, 404-6, 409-11, 414)	Error *Sticky Note*
Client-side Authorization (401-3)	Notify and disable the *sticklet*
Server-side Recoverable (500, 502-4)	Retry in a short period
Server-side Unrecoverable (501, 505)	Error *Sticky Note*
Credential Required (no code returned)	Error *Sticky Note*
"OSearch protocol" retrieves the empty result	Error *Sticky Note*
Brick description is not met	Error *Sticky Note*

Table 5.3: *Sticklet* built-in error handling stategies.

Fault Prevention

Fault prevention aims at preventing that faults are integrated into the system. Besides non-deliberate faults, we should also consider intentional threats such as creation of/redirection to phishing pages, stealing history information (or sensitive data stored on either pages or cookies), or port scanning upon the user's local network (refer to [Goo] for further details). The full expressiveness of *JavaScript*, its intricate coding and its interpreter-like nature make peering at the code of the script not an option. *Sticklet* abstracts away from some dangerous *JS* operations into more abstract and declarative description of the solution. In addition, *Sticklet* also ensures confluence for simultaneous *sticklet* enactments, avoiding the collision problem (see Subsection 2.3.4).

Fault Tolerance

Fault tolerance copes with the presence of faults. A system is fault tolerant if it can mask the presence of faults in the system by using redundancy. *Sticklet* hardwires basic handlers for a foreseeable, fixed set of cases. Table 5.3 indicates *Sticklet* recoverability strategies for the most common faulty

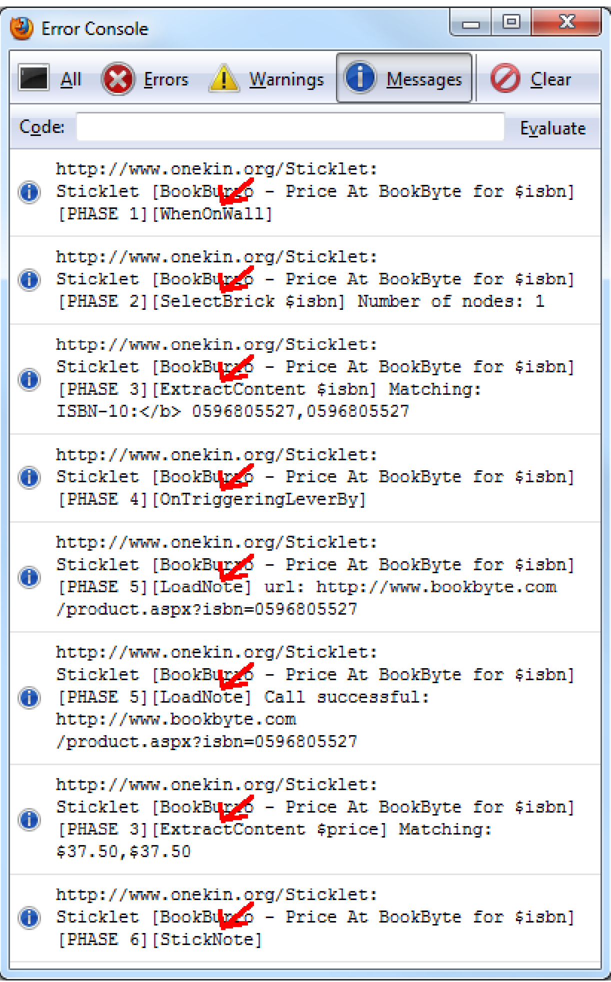

Figure 5.16: *Sticklet* tracing exemplified for *BookBurro*: *JavaScript* messages are abstracted into *Sticklet* terms (i.e. *WhenOnWall*, *SelectBrick*, etc.). Trace format: *"Sticklet [<projectName> - <stickletName>] [PHASE #][<phaseName>]"*.

scenarios. Besides *HTTP* errors, *Sticklet* also provides support for some *ad-hoc* defective situations, namely:

- the need for credentials when enacting *LoadNote*. In *BookBurro*, access to the library of the University of Manchester might require some credentials. The *HTTP* protocol already provides some codes to indicate this scenario. But, not all websites are so careful, and forget to inform the agent about this situation. That is, *Sticklet* can overlook this situation if only checks for the *HTTP* codes. This is the case for *the University of Manchester*. This scenario is frequent enough for *Sticklet* to be attentive and perform some term search in the returned page (e.g. “identification” or “account”) when the sought resource is not found in the returned page. If so happens, an error *note* is produced,

- the “osearch protocol” retrieves the empty result. It could happen that the current entity does not have a counterpart in the guest site (e.g. *Manchester Library* does not hold the book at hand). *Sticklet* just indicates this fact as an error *note*.

It could have been possible to enhance the *DSL*, and let users explicitly indicate a customized contingency action. Some examples follow: if the book is not available at the university library, try the city library; if *GoogleMaps* fails to retrieve a place then, query again by changing the order of parameters (might be latitude and longitude where wrongly placed in the host site). However, we did not experience a number of such scenarios large enough to ground the introduction of a new *DSL* construct.

Fault Removal

Fault removal aims at reducing the number of faults. This is related with testability which in turn, involves tracing and error reporting. *Sticklet* supports these two features. An important point is that tracing and error reporting should be conducted in *Sticklet* terms.

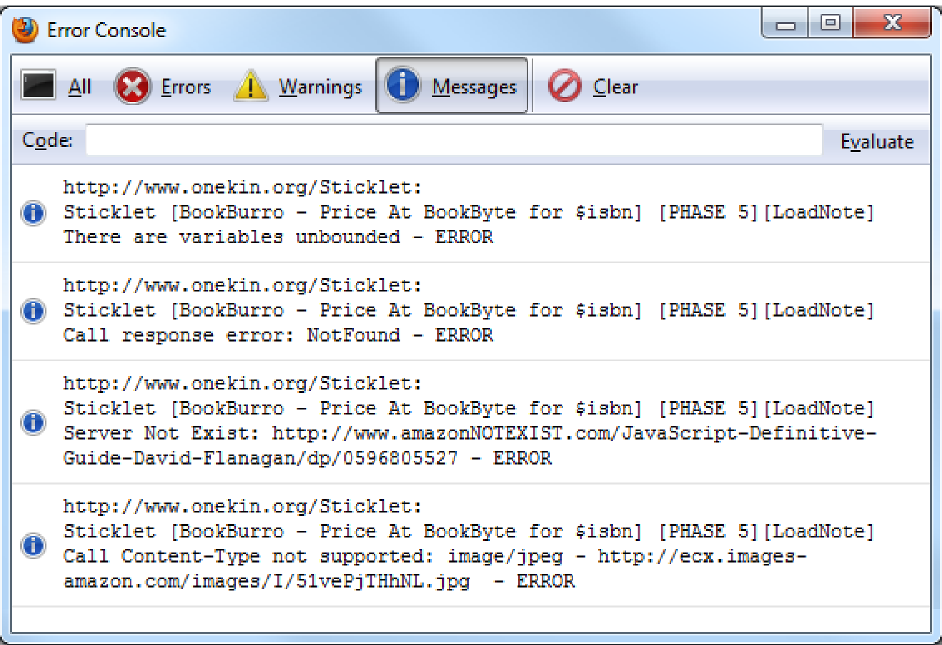

Figure 5.17: *Sticklet* error reporting. *Firefox* error console now displays *sticklet* errors: (a) matching variable not bounded, (b) call-response error, (c) server not found, and (d) content-type not supported.

Tracing. When developing scripts for others, debugging is a must. Tracing is a basic mechanism to follow and understand the flow of the scripts process. Since scripts are described in *Sticklet* terms, this flow is more abstract, and hence, easier to follow than its *JS* counterpart. *Sticklet* provides a trace message for each clause. Traces are displayed on the log console of the browser (for *Firefox*, press *CTRL+Shift+J*). Figure 5.16 provides an example while tracing the *BookBurro* script. Worth noticing: phase 2 reports the number of nodes fulfilling the match condition (i.e. this indicates how many times the *sticklets* are to be fired); phase 3 displays the content of *bricks* (hence the user is aware of which data is being extracted); phase 5 announces the *URL* to be called. In this way, consumers are aware of the data/services being used to achieve the augmentation, hence improving the trustworthiness on *sticklets*.

Error Reporting. *JavaScript* engines tend to be poor on error

reporting. Basically, developers are responsible for the handling of error states using "try/catch" blocks. This is so because the faulty cause could greatly differ among *JS* scripts. By contrast, *Sticklet* restricts the expressiveness of *JS*. This permits to limit faults to four scenarios: guest variable not bounded, call-response error, server not found, and content-type not supported (see Figure 5.17). In addition, *Sticklet* accounts for syntax checking and type checking[10].

5.5.2 Maintainability

We distinguish two rationales for *sticklet* maintenance. First, the evolution of the augmentation functionality. The augmentation functionality might need to be extended/reduced in either its scope or its content. This functionality is realized as a *stickletBox*, and its evolution will more likely imply addition/deletion of *sticklets*. Since *sticklets* are decoupled, additions/removals will have no impact on the remaining *sticklets* of the *stickletBox*.

The second rationale for maintenance is changes on the underlying website. This is more cumbersome since it might impact the layout, content or even *URL* structure that grounds the *sticklet*. Unfortunately, websites are reckoned to evolve frequently, with evidences in the figure of twice a year [DDSC07]. This may represent that the associated *sticklet* stops working twice a year. For only 5 or 6 *sticklets*, this rate might account for maintenance becoming a monthly burden. This is hardly bearable if directly conducted in *JavaScript*. This scenario calls for approaches where development is so straightforward that makes easier to develop from scratch than maintain. Besides declarativeness, *Sticklet* incorporates assistance mechanisms (see Subsection 5.6.2 *"No time to code"*) that greatly facilitate and speed development. According to first evidences (see Section 5.7), a *sticklet* takes around thirty minutes to develop. This limits

[10]Type checking is provided for the following types: *XPath, Regular Expression, URL Regular Expression, URL Call* and *String.*

to thirty minutes the maintenance penalty.

5.6 Web Augmentation: When Attention is Scarce

So far, we consider the bulk part of script *prosumption*: handling the code. However, this is not enough. We should also turn to other ancillary aspects of *prosumption* that could be negligible in an organizational setting, but become crucial in a Web2.0 scenario, namely, provisionability (i.e. facilitating prototyping), tailorability (i.e. easiness to customize the code), shareability (i.e. the facility to share an augmentation script), familiarity (i.e. to what extent the tool resembles what the users might be acquainted to), or operability (i.e. ability of the tool to be easily operated by a given user in a given environment).

Being full-fledged *GM* scripts, *sticklets* can tap into the tools available for *GM* script operation: editors, script repositories or installation utilities are also available to *sticklets*. However *GM* and *Sticklet* target different audiences: dedicated programmers *versus* occasional programmers. This difference impacts the operability requirements. Three clicks for editing a script could not be a big problem if you have the whole afternoon but it could be a nuisance if all you have is thirty minutes. We then reconsider *GM* operation at the light of three main tasks: coding (producer perspective) and, installation and sharing (consumer perspective).

5.6.1 Greasemonkey Operation

Traditionally, *Web Augmentation* is conducted by *JavaScript* programmers through weavers such as *Greasemonkey (GM)*. Although *GM* programmers are not our target audience, they know what augmentation is, and they can act as heralds of *Sticklet*. After all, *Sticklet* can serve for quick prototyping before moving to *JavaScript* for more sophisticated outcomes. Therefore,

we want *Sticklet* to sound familiar to *GM* users. This leads to a main decision: supporting *Sticklet* as an internal *DSL* of *JavaScript*[11] [McC07]. Indeed, *sticklet are installed and operated in the very same way as JS scripts in GM, i.e.:*

- *edited* as any other text file with extension "*scriptName.user.js",*
- *installed* by just dropping the *sticklet* file into the browser,
- *enabled/disabled* (i.e. temporarily stopping the augmentation) using *GM* facilities,
- *consulted* through the *GM* library that keeps the list of scripts currently installed,
- *shared* through the *userscripts.org* repository.

Our hope is to facilitate the transition of the 38 million *GM* users, the most active community of *Web Augmentation* to date. That said, *GM* targets programmers who can spend several hours developing their scripts, and consumers who do not hesitate peering at script repositories. However, this is not our scenario.

5.6.2 No Time to Code

New Scenario. Provisionability refers to the difference in production between professional coding and end-user coding [GBC+06]. When attention is scarce, users tend to be less systematic, commonly resorting to evolutionary and exploratory prototyping. We consider two likely coding scenarios:

- *impulsive prototyping.* We address short term and situational needs where programming is not a planned activity but a more circumstantial activity. For instance, a user can first augment

[11] This syntactic dependency from *JS* is surfaced in the use of the dot notation to concatenate the distinct *Sticklet* constructs.

Amazon with *BookByte* prices, get immediate feedback to see if it works, and differ to the next spare slot the completion of the script. This can be hours, days or, might be, the next time a purchase at *Amazon* reminds him about this *sticklet*.

- *opportunistic reuse.* The complexity and difficulty of programming can be reduced by giving end users a head-start with existing code, which they can adapt to their own purposes. Opportunistic reuse is regarded as a main enabler for end-user programming: "reuse is often what makes a project possible, since it may be easier for an end user to perform a task manually or not at all than to have to write it from scratch without other code to reuse" [Bla02]. This facilitates tailorability, i.e. consumers can customize third-party *sticklets* to their own preferences. In addition, users might be impelled to code when they try to mimic the augmentation achieved by someone else.

Tackling the new scenario. This way of working requires of agile means for enhancing and editing existing *sticklets*. Our aim is to make *sticklet* edition a more "impulsive" action so that editing can occur at the time and at the place where the augmentation takes place. Back to our first sample, the user reminds his one-*sticklet BookBurro* script at the time he is purchasing at *Amazon*. He wonders which could be the price at the *Powell*'s bookshop, and, at this very moment, he is impelled to enhance the script. The aim is to drive this impulse at the time it rises.

So far, the edition of *GM* scripts requires two clicks: one click on the *GM* icon at the status bar which opens a menu with the list of installed scripts; next, another click to select the script which makes the code to pop up in the default text editor. It is not a big burden (just two clicks) but you have to be determined to edit the script and move around the browser window to do those clicks.

Intended for general-purpose scripts, *GM* uses a general-purpose editing process. By contrast, *domain-specific languages* can benefit from domain-specific editors. And this refers not only to code the completion

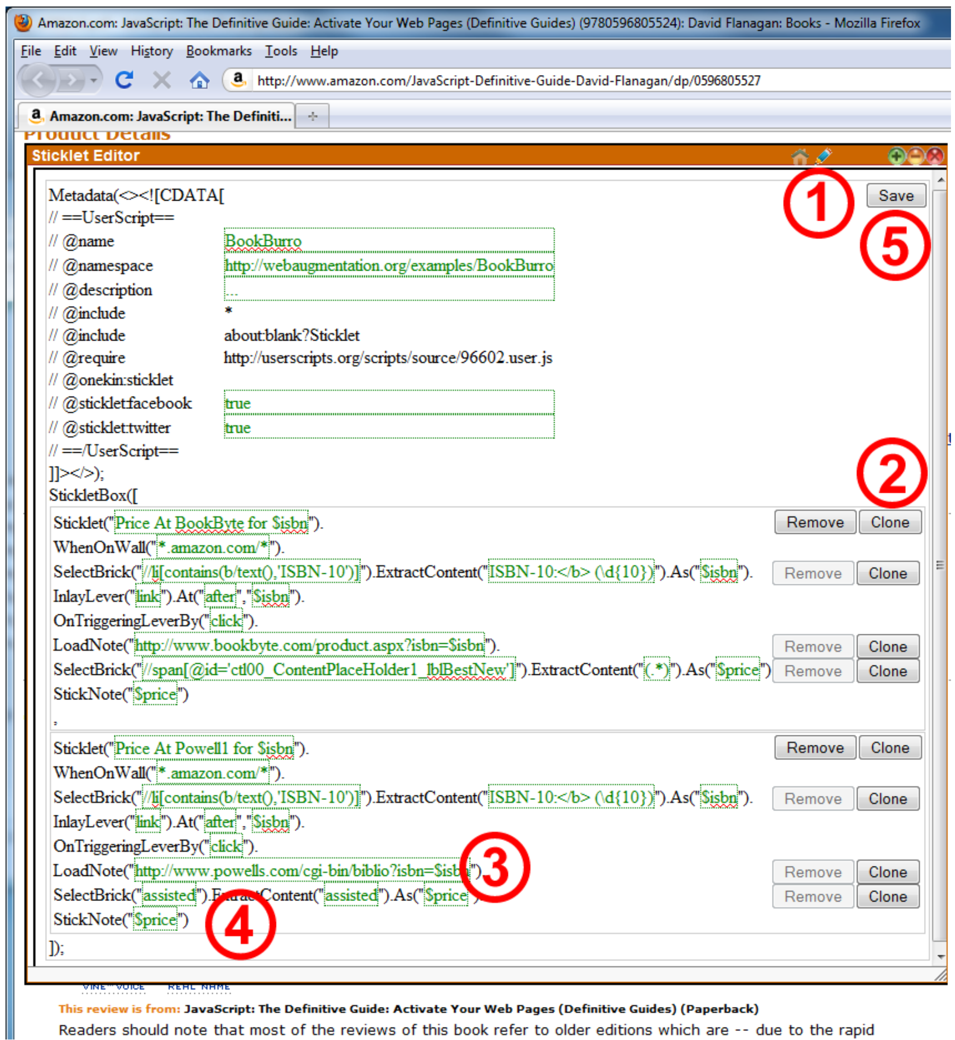

Figure 5.18: *Sticklet* inline editor.

feature but to contextualizing the editing process within the most likely scenarios of use. Both impulsive prototyping and opportunistic reuse call for edition to be integrated as part of the augmentation itself, i.e. you see the augmentation, you are impelled to edit the code.

In light of these considerations, *Sticklet* introduces an inline editor as part of the *sticklet* decorator (see Figure 5.18). Click on this icon and you can update, clone or delete the *sticklets*. Back to our example, the user can readily (1) edit *BookBurro* when at *Amazon*; (2) clone the *sticklet "Price At BookByte"*; (3) substitute the *LoadNote* clause with *Powell*'s URL, (4) assign "assisted" to the *XPath*-valued clauses, and (5), click the *"Save"* button. *Sticklet* will assist in setting the values for the unbounded clauses, and will regenerate the script. If you want to create a bright new script, just provide a different *"name"*. Otherwise, the *BookBurro* script is updated. The inline editor not only makes the code one-click away, but frames the edition within the most likely context of use: at the time the *sticklet* is enacted.

The bottom line is that

> **domain specificity not only impacts the constructs used to describe the solution but also the way to reach this solution. *JavaScript* and *Sticklet* not only differ in their primitives but also in their development processes.**

5.6.3 No Time to Install

New Scenario. *Greasemonkey (GM)* scripts are very easy to install: drop the *script* file into the browser and you are done. Since *sticklets* are functional *GM* scripts, so can be done for *sticklets*. However, this scenario assumes (1) *GM* is already installed, and (2), the script file is already in either the desktop or a remote repository. *Sticklet* departs from this setting by considering (1) consumers might not have *GM* installed, and (2), they may never hear about *Sticklet,* not even about *Firefox* add-ons.

Tackling the new scenario. This new scenario advices the installation

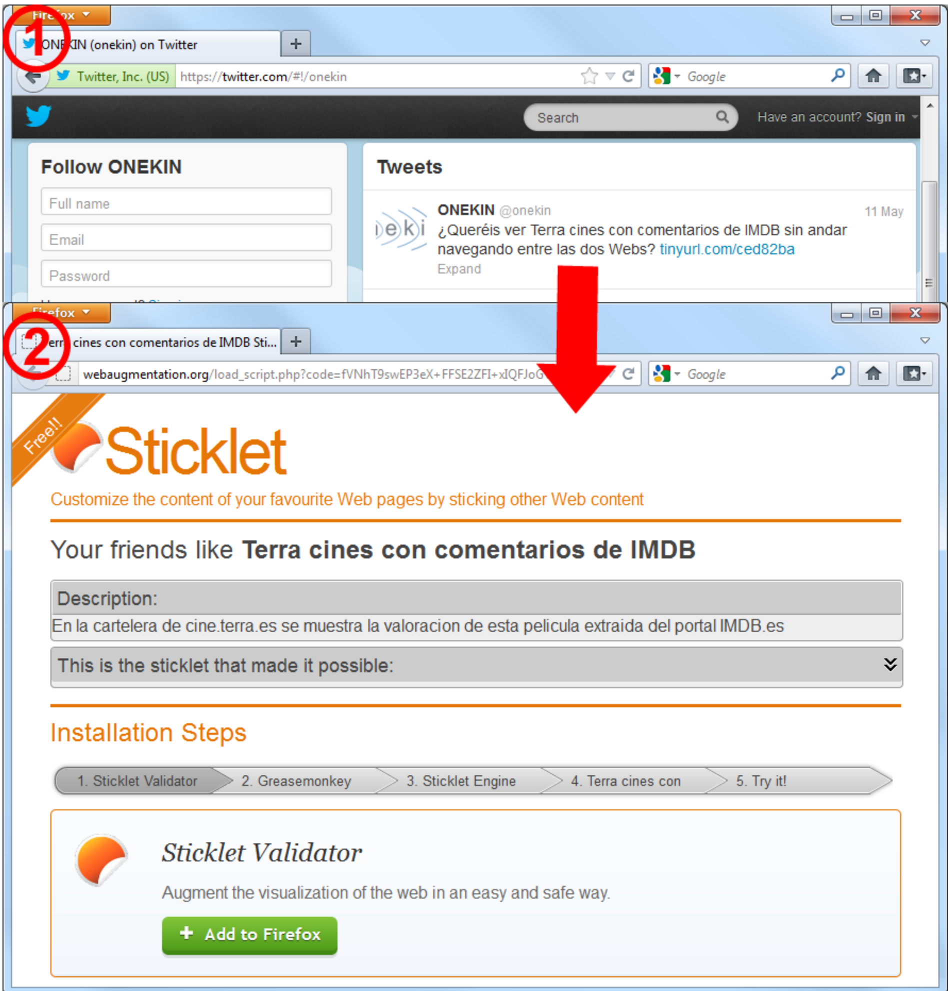

Figure 5.19: Installation of a *sticklet* from a tweet. The process is guided by a progress tracker.

to be somehow linked to the sharing process, better said, the sharing artefact. Fortunately, and unlike "desktop artefacts", "browser artefacts" (i.e. those enacted within the browser's boundaries) can download and deploy their interpreters at runtime as web services. The *Sticklet* engine is also available as a web service at *webaugmentation.org*. In this way, we introduce *"the sticklet URL"*, a *URL* that not only identifies univocally the *sticklet* but also denotes an implicit petition to the site *"webaugmentation.org"* to install this *sticklet*. This request is codified as a *tinyURL*. Note that the feasibility of this solution also rests on the compactness of the code: *DSLs* account for lean code which can be packed as a *URL*.

These *tinyURLs* can be obtained from the inline editor when clicking the *Twitter/Facebook* icon (see later). Next, you can share it through *Twitter* or just sent it through email. No matter the means, when clicking a *sticklet* URL, the *Sticklet* service cares for the burdens of your installation. The *Sticklet* service first verifies the receiver's browser configuration (i.e. *Sticklet* validator plug-in available, *Greasemonkey* plug-in available, *Sticklet* engine on), and next, installs the *sticklet* (provided user consent is granted).

Figure 5.19 shows the installation page that pops up when clicking a *sticklet URL*. The page contains a progress tracker that guides the user throughout the installation process. Depending on the current browser configuration, the starting point changes. Though this verification is conducted every time a *sticklet* is installed, the overhead is negligible while saving users from downloading themselves the *Sticklet* add-on. For instance, the *BookBurro tinyURL* is *http://tinyurl.com/cxw9ocy*. Just copy this *URL* in the browser bar to see this service at work.

In brief,

> **'browser artefacts' allow for installation to be a side-effect of sharing. When in a trusty setting, *URLs* can be used not only to univocally identify the artefact but transparently request the installation of this artefact.**

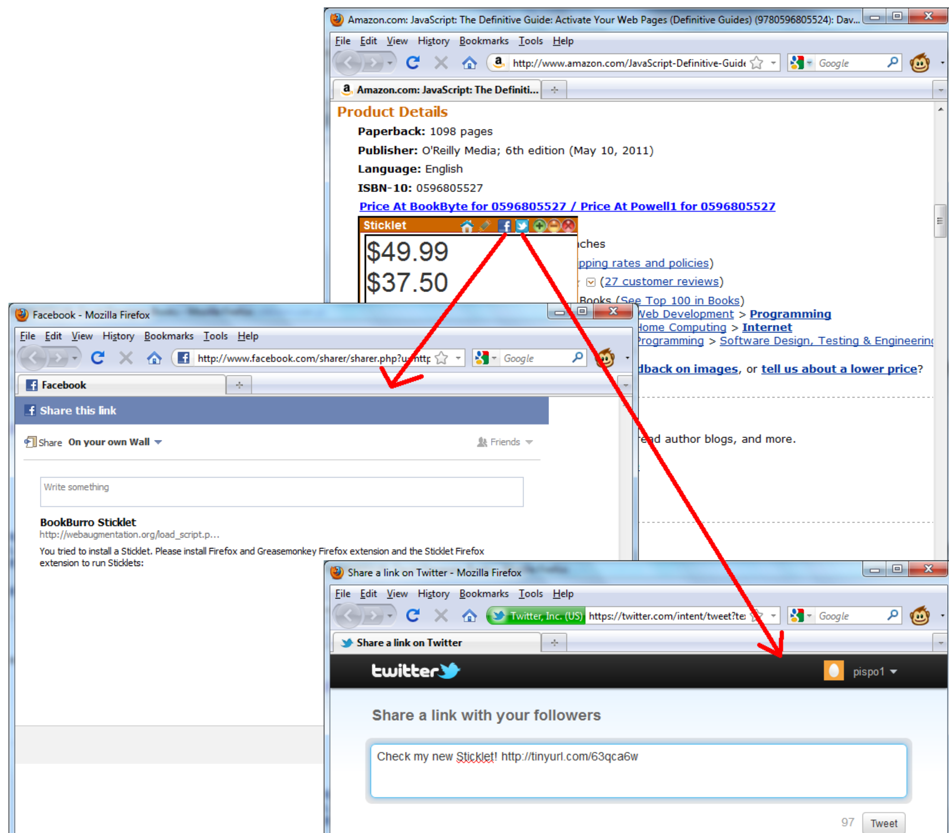

Figure 5.20: *Sticklet* inline sharing. Clicking on *Twitter* automatically generates a tweet which embeds the *sticklet tinyURL*.

5.6.4 No Time to Share

New Scenario. This paper starts with the vision of end users *prosuming* scripts as easily as they do for pictures or blog posts. One distinctive feature of this scenario, and key ingredient of the Web2.0, is sharing. Traditionally, script sharing is achieved through repositories (e.g. *Greasemonkey* and *Chickenfoot* follow this approach). Being valid *JavaScript* code, *sticklets* can be uploaded and managed through *Greasemonkey* repositories. Indeed, you can find several *sticklets* at *http://userscripts.org/users/sticklet*. On the upside, repositories provide public access to not only the code but also the reviews or discussions about the script. On the downside, repositories force producers to disclose

their scripts to the wider public which could be intimidating for end users. Indeed, studies from social networks indicate an increase in sharing if conducted within smaller groups [BBCS08].

Tackling the new scenario. *Sticklet* explores a new scenario where consumption is reactive (i.e. someone tells you about the *sticklet*) rather than proactive (i.e. you looking into a script repository). The aim is to let users share *sticklets* without the burden of uploading them in a repository. Like pictures. After all, sharing pictures does not demand uploading them first at *Flickr*. We explore "inline sharing" whereby *sticklets* themselves offer the means to be shared through the social networks. Specifically, *Sticklet* syntax includes two annotations: @*sticklet:facebook* and @*sticklet:twitter* (see Figure 5.20). These boolean annotations permit the producer to indicate whether the script can be shared through *Facebook* or *Twitter*, respectively. These annotations make the *sticklet* decorator exhibit the icons of these popular sites. By clicking the *Twitter* icon, users create a tweet that includes the *sticklet*'s *tinyURL*.

The bottom line is that

> **sharing "should be made as simple as possible but not simpler" as a way to foster production.**

5.7 Evaluation

The main goal of *Sticklet* is to make *Web Augmentation* accessible to consumers and producers alike. In this sense, the matter is mostly about affordance, i.e., making augmentation accessible to a wider audience. In this setting, the quality of use becomes paramount, i.e. "the user's view of the quality of a system containing software, and is measured in terms of the result of using the software, rather than properties of the software itself" [ISO01]. *ISO-9126* provides a framework to evaluate quality in use. This section provides a preliminary evaluation of *Sticklet* along the *ISO-9126*'s quality-in-use dimensions: *(i) effectiveness* (i.e., the capability of the

software product to enable users to achieve specified goals with accuracy and completeness), *(ii) productivity* (i.e., the capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness), *(iii) safety* (i.e., the capability of the software product to achieve acceptable levels of risk), and *(iv) satisfaction* (i.e., the capability of the software product to satisfy users). Effectiveness and productivity were measured objectively: number of completed tasks and minutes to complete them, respectively. On the other hand, while safety is an objective measure, our target population (i.e., people with no previous knowledge of *JavaScript*) may not have the means to perform such evaluation. Hence, we opted for evaluating trustworthiness, how trustworthy participants perceived *Sticklet* is. Both trustworthiness and satisfaction were assessed through specifically designed questionnaires.

At the time of this writing, *Sticklet* has been available as a *Mozilla* add-on for over a year. According to the *Mozilla* figures[12], *Sticklet* has 37 average daily users: 20% are in Spanish (which can be anticipated by the nationality of the authors); 30%, 8% and 7% are in English, German and Italian, respectively (which can be explained by the presentation of *Sticklet* at a demo session of an European conference); finally, 35% come from other languages (Chinese, Russian, etc.) which can be attributable to serendipity searching. These numbers are still very low to permit a proper evaluation of *Sticklet* "in the wild". Hence, this evaluation is based on subjects mainly taken from academia.

Sticklet design is driven by satisfying producers and consumers alike. Nevertheless, these two types of users have different requirements (e.g., expressiveness vs. trustworthiness) and backgrounds (hobby programmers vs. computer literates). Thus, each evaluation targets different audiences.

[12] *https://addons.mozilla.org/addon/Sticklet/statistics/?last=90*

5.7.1 Sticklet consumption for computer literates

A comparative design was adopted to examine the influence of distinct background variables in the perception and use of *Sticklet*. The sample was composed of a group of users with similar characteristics of the potential consumers of *Sticklet*.

Research Method

Setting. In order to eliminate differences in the perception of *Sticklet* due to hardware or bandwidth differences, the study was conducted in a laboratory of the *Computer Science Faculty* of *San Sebastian*. All participants used computers with the same features (i.e., *Intel Core 2 1.86 GHz*, *3 GB RAM* and *Windows XP Professional SP3*) and a clean installation of *Firefox* 12.0.

Procedure. The study intended to mimic as closely as possible the circumstances potential consumers might encounter when deciding to try a *sticklet* script. Hence, explanations and instructions were reduced to the minimum. Before the participants started, they were informed about the purpose of the study and were given a brief description (5 minutes). The sample *sticklet* augments the movie listings page at a popular Spanish portal, *cine.terra.es/cartelera*, with scores and comments from the website *www.imdb*.es[13]. Next, participants were handed out a sheet that instructed them to access to the *Onekin Group* twitter page[14] where a tweet, supposedly sent by a mate, commented about the wonders of a *sticklet* he had made, and provided this *sticklet*'s *tinyURL*. From then on, *Sticklet* assisted participants through a progress tracker (see Figure 5.19). In order to measure productivity, participants were then asked to note down the time when they clicked the link in the tweet, and again after installation, when they were able *to see* the augmented *Terra* page. Last,

[13]This *sticklet* can be downloaded from *http://tinyurl.com/ced82ba*.
[14]*http://twitter.com/onekin*

participants were directed to a *GoogleDocs* online questionnaire to gather their opinion about *Sticklet*.

Subjects. Thirty three first year undergraduate computer science students participated in the study. The majority of participants were male (78.8%). Regarding age, 81.8% were in the 18-20 age range and all participants were below thirty years old. 63.6% check movie listings before watching a film but only 6.1% were already familiar with the *Terra* website. 90% access social networks on a daily basis and 45.5% tweet with a weekly frequency. Last, concerning the participants' browsing behaviour, in the last year participants installed between 0 and 20 applications/plugins/add-ons, with a mean of 5.4.

Instrument. An online questionnaire served to gather users' experience using *Sticklet*. It consisted of five parts, one to gather the participants' background and one for each of the quality-in-use dimensions. In order to evaluate effectiveness, the questionnaire contained the proposed tasks so that participants could indicate if they had performed them, while productivity was measured using the minutes taken in such tasks. Trustworthiness and satisfaction were measured using 7 and 10 questions, respectively, using a 5-point Likert scale (1=completely disagree, 5=completely agree).

Data Analysis. Descriptive statistics were used to characterize the sample and to evaluate the participants' experience using *Sticklet*. Moreover, t-test analyses were performed to assess differences among groups of users (e.g., familiarized vs. non-familiarized with film listing websites). *PASW Statistics 18 for Windows*[15] was employed to perform the different analyses.

Results

We begin by describing the qualitative evaluation. Two observers were present during the evaluation, which were instructed not to interfere unless

[15] *http://www.spss.com.hk/statistics/*

strictly necessary. Only 2 out of 33 participants had doubts during installation and only in one case was the intervention of one of the observers necessary. This was due to the user leaving the page before accepting a pop-up window that prompts the user to accept the installation of the *sticklet*[16]. Five of the participants failed the exercise since they forgot to reload the page to see the augmentation at work. That is, since the exercise began in the *Terra* page, and next, they were instructed to download the *sticklet*, these participants expected the *Terra* page to change all of a sudden as part of the script installation. Despite being so instructed, these participants missed to reload the page, and hence, failed to see the augmentation.

Besides the questionnaires for the quantitative evaluation (see below), an optional open question was left to gather the participant's comments. Eighteen participants chose to give their impression of the tool. One participant found *Sticklet* too convoluted for her needs. She argued that the browser's tabs were sufficient to keep different sites open simultaneously, despite the cognitive fragmentation that goes in moving between tabs. Otherwise, the perception was positive. Only one participant found the installation process cumbersome. Three participants suggested the *Sticklet* decorator to resize automatically to fit the *sticklet* output. One participant had the impression that the sharing of a *sticklet,* even from an acquaintance, does not guarantee that the *sticklet* is virus free since "tweeting is not a serious way to share code". The point to note here is that the trustworthiness of the object (that being some data or code) is that of the informer. All in all, this observation introduces the conduit as a source of trust. Another noteworthy happening is that two subjects were playing around with the *Sticklet* inline editor, though no hint about *Sticklet* editing facilities was given to the participants. This suggests that incorporating the inline editor as part of the *Sticklet* decorator might encourage consumers to play around, and eventually, become producers. By permitting editing

[16]To prevent this from happening in the future, the installation process was later modified to wait for the pop-up (modal window) before continuing the process.

Task	Frequency	%
Complete installation	33	100
Access *Terra* website and see augmentation	28	84.8

Table 5.4: Effectiveness Results.

from the very same place where the output is rendered (no need to deploy additional browser menus), the *sticklet* code is just one-click away from its visible effects. Next, the quantitative evaluation of the dimensions described above is detailed.

Effectiveness

Effectiveness is the capability of the software product to enable users to achieve specified goals with accuracy and completeness [ISO01]. For consumers, effectiveness has to do with installation and enactment of *sticklets*. The questionnaire asked for these two tasks (see Table 5.4). All participants were able to complete the installation, i.e., install *Greasemonkey*, *Sticklet* and the *sample sticklet*. Interestingly enough, five participants did not catch the need to reload the original page for the augmentation to show up, but expected this process to take place automatically.

Productivity

Productivity is the capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness [ISO01]. We asked participants to note down the time first when they clicked on the tweet's *tinyURL* and again when they accessed the augmented page, after installation. Productivity was collected by requesting the number of minutes between the two moments. Participants reported between 2 and 10 minutes, with a mean of 3.22 minutes, from the moment they clicked on the tweet to the point where they accessed the augmented page.

	Computer Literates		Hobby Programmers	
Item	**Mean**	**St. Dev.**	**Mean**	**St. Dev.**
1. In general the demo has inspired me confidence	3.73	1.126	3.89	0.9362
2. The fact that *Sticklet* is downloaded from the official *Firefox* plugin page inspires me confidence	4.24	1.091	4.34	0.7454
3. To know that a friend has developed the *sticklet* would inspire me confidence	4.12	1.111	4	0.9428
4. To know that a friend has developed the *sticklet* and has shared it on *Facebook* would inspire me confidence	4.09	1.128	3.78	1.0304
5. The fact that I understood the code inspires me confidence	3.79	0.857	4.28	0.8031
6. To know that *Sticklet* does not allow malicious *JavaScript* code would inspire me confidence	4.27	0.977	4.27	0.977
7. To know that *Sticklet* was recommended by a friend via *Twitter*, even if she/he was not the developer, would inspire me confidence	3.03	1.212	2.73	1.3664

Table 5.5: Trustworthiness Perception Results from 1 (total disagreement) to 5 (total agreement).

Trustworthiness

This section intends to measure trustworthiness *perception* of participants (i.e. whether *sticklets*' behaviour is perceived to be consistent, predictable and trustworthy). Table 5.5 summarizes the results.

Items 2 and 3 reveal how trust on both *Firefox* (as the container of plugs-ins) and friends (as the authors of *sticklets*) are perceived as the strongest source of confidence. It is interesting to observe that item 4, i.e. the fact that friends publicize their *sticklets* via *Facebook* (as a sign of pride in the *sticklet*), seems to add no extra confidence to consumers when compared with being developed by a friend (i.e., item 3). Both the understanding of the *Sticklet* code (item 5) and its built-in security rules

(item 6) are also highly regarded. Notice how code understanding (item 5) is the one with lower standard deviation (i.e. 0.857) which suggests the highest unanimity as a confidence builder. Finally, sharing via *Twitter* (item 7) is ranked lowest as a confidence builder. This might suggest that the notion of "friend" in *Twitter* is diluted to merely mean list of contacts rather than true friends you can rely upon. This seems to indicate that *Sticklet* online tweeting did not scale to our expectations as a trust-builder.

Relations among variable differences among groups, according to background variables, were analysed using t-test. We were specially concerned about the impact that being familiarized with film listing websites could have in the perception of the *sticklet*, so that regular users of those sites could have measured up *Sticklet*. However, we found no statistically significant differences in either trustworthiness related to being an official *Firefox* plugin, trustworthiness related to being produced by a friend, or trustworthiness related to understanding the code and its built-in security rules between participants familiarized with film listing websites and those who were not. Where we did find statistically significant differences was among the participants that had installed few (two or less) and multiple (three or more) applications/add-ons/plugins in the last year. Specifically, those who installed multiple apps showed higher confidence related to being an official *Firefox* plugin (Mean=4.63) than those who installed less (Mean=3.92, p=0.033). The same happened when considering trustworthiness related to being produced by a friend (Mean=4.58 vs. 3.69, p=0.009) and trustworthiness related to the built-in security rules (i.e., not allowing malicious *JavaScript* code) (Mean=4.63 vs. 4.00, p=0.024).

Though consumption was mainly targeted to computer literates, we were also interested in analysing the impact that a more technical background has on trustworthiness perception. To this end, we handled the same questionnaire to the "hobby programmers" group (see next subsection). Results can be seen in Table 5.5. It is interesting to observe how confidence by code understanding (item 5) raised to a mean of

Item	Mean	St. Dev.
1. It is comfortable to be able to click on a tweet to install the tool	4.21	0.857
2. During the installation process, I always knew what I was meant to do	4.48	0.795
3. The companion *Twitter* message helped me understand what the *sticklet* did	3.52	1.064
4. The *sticklet* code helped me understand what the *sticklet* did	3.42	1.001
5. Looking at the final augmentation outcome helped me understand what the *sticklet* did	4.06	1.088
6. There were no errors during the installation and execution of the *sticklet*	4.45	1.201
7. The *Sticklet* engine is fast enough	4.33	0.890
8. I think augmenting a website with data from another website is a good idea	4.27	1.008
9. *Sticklet* saves me time, since I don't have to browse through different webs	4.24	0.902
10. I found the demo interesting enough to share it with my friends	3.42	1.001

Table 5.6: Satisfaction Results from 1 (total disagreement) to 5 (total agreement).

4.28. No statistically significant differences were found in trustworthiness perception between computer literates and hobby programmers.

Satisfaction

Satisfaction is the capability of the software product to satisfy its users [ISO01]. Satisfaction can be measured along three dimensions: perceived usefulness, perceived ease of use and willingness to use in the future [Dav89]. In this case, the product is the *sticklet* script, but our interest does not lie in measuring the usefulness of a specific *sticklet* but on what aspects of the *sticklet* script are derivable from the *Sticklet* engine. Therefore, we do not measure the "perceived usefulness" of this particular *sticklet*. Rather, we investigate which aspects of the *sticklet* script can be traced back to features of the *Sticklet* engine i.e. those that can be applied to any *sticklet* script no matter its function (e.g. efficiency, understandability,

soundness, etc.). The list of items and the results are summarized in Table 5.6.

As in the case of trustworthiness, being familiarized with the website did not establish statistically significant differences in any of the satisfaction items. Regarding app installation behaviour, statistically significant differences were found in items 2 to 5, suggesting that participants that install apps more often had a better understanding of both the installation process and what the *sticklet* did (item 2, Mean=4.7 vs. 4.15, p=0.029; item 3, Mean=3.8 vs. 3.07, p=0.0220; item 4, Mean=3.8 vs. 2.85, p=0.0030; item 5, Mean=4.4 vs. 2.54, p=0.0186). As for the use of *Twitter*, a significant difference was found between participants that tweet at least once a week and those who tweet monthly or never in item 1 (i.e., comfort of installation clicking on a tweet, Mean=4.53 vs. 3.54, p=0.0303).

5.7.2 Sticklet production for hobby programmers

A comparative design was adopted again to examine the influence of distinct background variables in the perception and use of the tool. A sample of users with more technical qualification was sought to evaluate the perception of potential *Sticklet* producers. In this case, a call to participate in the study was issued among the faculty and Ph.D. students of the *Computer Science Faculty* of *San Sebastian*. The evaluation also included questions about background in related technologies (e.g. *JavaScript*, *XPath*, etc.).

Research Method

Setting. The study was conducted in the same laboratory the consumer study had taken place.

Procedure. All participants were handed out a sheet with instructions for each task (e.g., what Web to access, when to take note of the time,

etc.). The study was divided in three tasks. Before they started, a general description of Web Augmentation was given (15 minutes).

First task: consuming a *sticklet*. It was exactly the same the consumers had performed, i.e., installation of *Sticklet* and the execution of a particular *sticklet* starting with a tweet. In this case, the *BookByte* online bookshop[17] was augmented with the price for the same book in *Powell's Books*[18].

Second task: modifying a *sticklet*. A brief description about each clause of the language was provided, with an emphasis on the *assisted* option (15 minutes). This option was shown at work for the *Powell's Books sticklet*. For the second task participants repeated what they had seen and regenerated the script using the assisted option to obtain the same functionality.

Third task: cloning a *sticklet*. At the beginning of the task a description of the *osearch* option was provided, using the book prices at *Walmart*[19] as example (15 minutes). Then, participants augmented the *BookByte* website with the reviews and ratings found in *GoodReads*[20]. To perform this task participants had to use both the *assisted* and *osearch* options, as well as searching the book by title (so introducing a new *brick* since the original *sticklet* searched by ISBN). Finally, participants were directed to a *GoogleDocs* online questionnaire to gather their opinion about *Sticklet*.

Subjects. Eighteen faculty and Ph.D. students participated. The majority were male (61%) and the mean age was 33.3. Concerning browsing behaviour, in the last year participants installed between 0 and 15 applications/plugins/add-ons, with a mean of 5.5. 61% access social networks on a daily basis while 66% never tweet. We also gathered information about their background on related technologies (e.g., *JavaScript*) using a 5-point Likert scale (1=none, 5=expert). Participants reported a mean of 2.1 for *JavaScript* knowledge and 33% had no previous knowledge. As for *XPath*, a mean of 2.1 was reported. Regarding the

[17] *http://www.bookbyte.com/*
[18] *http://www.powells.com/*
[19] *http://www.walmart.com/*
[20] *http://www.goodreads.com/*

example, 50% buy books online and 66% check online reviews before buying a book. Last, only 11% had accessed *BookByte* before the study.

Instrument. The questionnaire consisted of four parts: background, effectiveness, productivity and satisfaction. Production does not involve any major risk. However, as we wanted to measure the difference between both groups of participants on trustworthiness perception, the same questionnaire on this issue was provided also in this case (see above).

Data Analysis. Same that those for the consumers case.

Results

First the qualitative evaluation is provided. Following the same design we used in the previous evaluation, the same two observers were present in this case. The first task was successfully completed by all participants with the only help of the previously provided explanation (15 minutes for each task). In the second task (i.e., modification of the provided *sticklet*), three participants encountered difficulties when filling the regular expression of the *ExtractContent* clause using the *assisted* option. An observer intervened to help these participants with the regular expression, to allow them proceed with the task. Four of the participants were not able to finish the third task, failing to understand the semantics of *osearch*.

Besides the questionnaires (see below), an open question was included. From the 18 participants, 13 commented. The opinions about *Sticklet* were positive. Some participants provided suggestions to improve *Sticklet* in the future. One participant suggested to shorten the installation process by eliminating some of the steps. Another cared about browser interoperability i.e. whether *sticklets* can run in browsers other than *Firefox*, which so far it is not the case. Four participants commented on the *assisted* option being useful when building *sticklets*. Two indicated that it would be nice to have the *assisted* option for all *Sticklet* clauses (e.g., *LoadNote*), thus eliminating the need to write the expression manually. One more subject aligned to this thesis by suggesting that the *assisted*

Task	**Frequency**	**%**
Task 1: Installation and see augmentation on the *BookByte* website	18	100
Task 2: Modify the *sticklet* using the *assisted* option	18	100
Task 3: Create new *sticklet* to see *GoodReads* comments	14	77.78

Table 5.7: Effectiveness Results.

option should be the default mode of dealing with *Sticklet*, thus shielding the user from the syntax. This is certainly an interesting follow-on: providing a wizard that guides the user throughout the development of the whole *sticklet*. Next, we look at the quantitative evaluation.

Effectiveness

Table 5.7 provides the fulfilment for the three tasks (i.e. installation, modification and cloning to create a new *sticklet*). Only the last presented some difficulties, where four people were not able to create the new *sticklet* in the allocated time (30 minutes). The elapsed time is based on a previous study were participants required 21 minutes on average to develop a new *sticklet*.

Productivity

Productivity is measured as the number of minutes required for each task: installation took between 2 and 7 minutes, with a mean of 3.39 minutes; modification took between 2 and 8 minutes, with a mean of 3.16 minutes, and finally, cloning (only for those that successfully completed the task) required between 8 and 24 minutes, with a mean of 14.5.

Satisfaction

Satisfaction is the capability of the software product to satisfy its users [ISO01]. In this case, the product is the *Sticklet* engine, and its ability to develop a working *sticklet*. While consumers focus on the usefulness of a specific *sticklet*, producers take a step back and evaluate whether

Items	**Mean**	**St. Dev.**
1. I think augmenting a website with data from another website is a good idea	4.62	0.6781
2. I think *Sticklet* is useful for avoiding going back and forth between websites	4.11	0.8089
3. I think *Sticklet* helps to keep focus without being distracted by browsing	3.95	1.1772
4. I think *Sticklet* is useful for decreasing the comfort threshold to recover data for decision taking (e.g., which book to buy)	4	0.8164
5. There were no errors during the installation and execution of the *sticklet*	3.72	1.2385
6. I think *Sticklet* in-line editor is easy to use	3.78	0.7857
7. I think *Sticklet* in-line editor eases clone&own	4.33	0.6667
8. I think *Sticklet* in-line sharing facilities (e.g., *Twitter* button) impulse sharing	3.33	1.1055
9. I think the *Sticklet assisted* option and its intersperse grid are easy to use	4.44	0.8315
10. I think it is easy for me to develop *sticklets* using the *assisted* option	4.5	0.6872
11. I would like to install other user's *sticklets* in the future	3.72	0.9313
12. I would like to exchange *sticklets* in the future	3.67	0.8819
13. I would like to keep developing *sticklets* in the future	3.94	1.0259

Table 5.8: Satisfaction Results from 1 (total disagreement) to 5 (total agreement).

augmentation itself paysoff for the effort to learn the *Sticklet* language. Along the lines of the approach proposed in [Dav89], we first evaluated the perceived usefulness of augmentation itself (items 1-4). Next, we focus on *Sticklet* as a mean to obtain the augmentation end. Specifically, the usefulness of *Sticklet*'s inline editor, the *assisted* option, and the inline sharing facilities (items 5-10) were measured. Finally, we want also to measure the willingness to use *Sticklet* in the future (items 11-13). Results are summarized in Table 5.8.

We found no statistically significant differences in any of the items between participants that reported previous expertise (3 or higher) or no

expertise (2 or lower) in *JavaScript*. In the case of *XPath*, item 10 (i.e., ease of developing using the *assisted* option, Mean=4.87 vs. 4.2, p=0.0154) was higher valued by *XPath* knowledgeable users. This seems to suggest that participants with previous knowledge of *XPath* were able to appreciate the advantages of the *assisted* option as opposed to having to write the *XPath* expression by hand. Statistically significant differences were also found among the participants that had installed few and multiple applications/add-ons/plugins in the last year. Participants that install more believed that *Sticklet* is more useful (item 2, Mean=4.42 vs. 3.5, p=0.062). Moreover, participants that install more ranked higher the questions related to the *Sticklet* inline editor (item 6, Mean=4.08 vs. 3.17, p=0.0026; item 7, Mean=4.58 vs. 3.83, p=0.0309), which may be due to being more accustomed to trying new things on the web.

5.8 Discussion

There is not a universal, one-size-fits-all *Web Augmentation* tool. It much depends on the task and target audience. *Sticklet* most distinctive feature is to move consumer requirements to the forefront. This implies to care for trustworthiness but also shareability, familiarity or installability, which are commonly overlooked in other works. In the following subsections, the requirements are put in the *Sticklet* context.

5.8.1 Expressiveness

DSLs imply to find a compromise between expressivity and generality. *Sticklet* as *JavaScript* internal DSL, cannot pretend the generality of *JavaScript* where any kind of modification is possible (i.e. content, colours, fonts, layout, can all be modified). Rather, we focus on a common kind of augmentation: *content-based augmentation based on third-party services*. Therefore, *our DSL* cannot delete content (only additions are permitted) neither can it change the layout of the target

page. However, we have evaluated *Sticklet* with non-trivial examples to validate its expressiveness. Compared with the rest of the works, we can say that the expressiveness of *Sticklet* is **medium** because it is not restricted to a concrete type of sites like *ActiveTags*. However, the changes allowed are restricted in comparison with *Chickenfoot* that allows to modify the webpage in anyway. This characteristic is directly related with the following one, learnability.

5.8.2 Learnability

Learnability and expressiveness are both intimately related, improve the learnability is at the expenses of expressiveness reduction. In *Sticklet*, we found a balance between them, *JavaScript* expressiveness is reduced in order to make it easier to learn. We used *Web Augmentation* domain constructs in the language, trying to make it as close as possible to the terms used by the target audience. *Sticklet* is more difficult to learn than *Platypus* because the visual support in our tool is limited. Nevertheless, a *Sticklet* expression is fixed by a small set of fixed constructs compared with *MashMaker*. We qualify the learnability of *Sticklet* as **high**.

5.8.3 Trustworthiness

Trustworthiness is a key characteristic in an informal scenario, if the users are not forced to use a tool but it fails and there is no mechanism to solve it, they will stop to use it. We improved trustworthiness is three ways: fault prevention, fault tolerance and fault removal. This work is trustworthy because we validate the expressions generated, in contrast with *Chickenfoot* that allows all the *JavaScript* statements. We can say that *IE Accelerator* and *Sticklet* are similar because the constructs are domain specific so there is no option to create a malicious artefact, so the *Sticklet*'s trustworthiness is **high** .

5.8.4 Maintainability

Maintainability is impacted by code readability as well as modularization. The use of domain-specific terms certainly made augmentation scripts more readable making it **highly** maintainable. In addition, *stickletBoxes* are modularized in terms of *sticklets*. Being self-contained, *sticklets* can be easily added/deleted with no impact on the other *sticklets* of the *stickletBox*. The rest of the works are not as maintainable as ours because they are not modularized or the modules are greater than ours.

5.8.5 Understandability

Sticklet's understandability is inherited from *Domain Specific Languages (DSL)*, therefore its understandability is **high**. *DSLs* are not a substitute for programmers but a way to increase their productivity while improving the reliability and understandability of software. We improved the understanding of *SelectBrick* and *ExtractContent* clauses with an explanation of their content via automatic code comments. The closest works in terms of understandibility to *Sticklet* are *Chickenfoot* and *IE Accelerator* because they make use of domain abstractions.

5.8.6 Tailorability

Tailorability describes the capacity of the system to be customized or adapted. Usually, tailorability is supported by the application itself; the developer of the application have this requirement in mind and add extra code to handle this situation. *MashMaker* makes the difference is this way, the customization is supported via *widget* properties. We say that *Sticklet* is **highly** tailorable because expressions are self-contained and its generation are simplified using the *assisted* feature.

5.8.7 Operability

Operability refers to intuitiveness in the operation of the tool. We considered the operability of the tool taking *Greasemonkey* operations as reference: the creation, edition, installation and sharing of the functionality. We improved the previous operations minimizing the number of clicks and hence, simplifying the process of each operation. *Sticklet* is mainly textual, compared to visual solution of *Platypus* that facilitates the creation of scripts. This is the main reason to say that *Sticklet*'s operability is **medium**. Moreover we provide debug and tracing facilities that eases the use of the tool.

5.8.8 Provisionability

Provisionability refers to the difference in production between professional coding and end-user coding. We simplified the effort needed to create and modify *sticklets* by introducing and editor that is a click-away from the sticklet execution and a clone/own function. We envision these mechanisms as helpers to migrate the users from consumers to producers. The functionality of rest of the solutions are coarse-grained so they are more difficult to create or reuse than ours, leaving our provisionability ranked as **medium**.

5.8.9 Installability & Shareability

Installability refers to the effort needed to add existing functionality to the system. This characteristic is strongly related with the shareability of the existing functionality. In order to install new functionality, first it is needed to be obtained possibly by being shared by others. The effort needed to install *sticklets* is that of *Greasemonkey*, only 1 click, because our solution is built on top of it, so its installability is **high**. For the sharing point of view, we improved the *Greasemonkey* sharing mechanism based on repositories. We permit to store the content of a *sticklet* in a *URL*. Being

possible to store a *sticklet* in a *URL*, now it can be shared with your mates through social networks. It is no longer necessary to convince your friends to create and account in a new website but only to install the *sticklet* engine and enjoy *Web Augmentation*. We conclude this section asserting that the shareability of sticklets is **high**.

5.9 Conclusions

We introduced *Sticklet*, a textual *DSL* for *Web Augmentation* targeted to end users. *Sticklet* is based on *JavaScript* but limits *JavaScript* generality for the sake of learnability and trustworthiness. Learnability wise, "end-userness" is pursued by limiting the set of constructs, and hardwiring a collection of heuristics in the engine. Based on a range of previous works, heuristics shelter users from the intricacies of (1) generalizing *XPath* expressions, (2) rendering of *XML/JSON* documents, (3) rendering of *Sticklet notes* to *HTML*, and (4) entity linkage using *OpenSearch*. As a result, *Sticklet* aims not only at reducing the learning curve but also bringing readability and declarativeness, and in so doing, improving trustworthiness. In addition, *Sticklet* tackles consumer concerns in a scenario of stingy attention: provisionability, familiarity, shareability or operability have played an important role in designing *Sticklet*. Some of these concerns are rather new but we believe will play an increasing important role as Web2.0 practices spread along.

Chapter 6

Conclusions

6.1 Overview

Web Personalization has proven to be an adequate paradigm to improve the satisfaction of web customers. However, the chances of customers and indirect content providers to actively participate in this process are limited. This dissertation addresses *Open Personalization*, a set of architectures that allows third parties and customers to customize websites.

In this chapter the main results are reviewed, their limitations are stated and new areas for future research are suggested.

6.2 Results

This dissertation extends the notion of Web Personalization by opening it to different actors. Based on the involvement of such actors, different scenarios are considered. Whereas Chapters 3 and 4 faces the personalization made by third parties, Chapter 5 allows end users to personalize the web. Specifically:

- *Chapter* 3 allows partners to personalize the company's website (*Server-Side Open Personalization*). An architecture of participation is proposed, based on the usage of an interface for web modification. The company defines the interface whereas the partners build on top of that. Using such interface, the company controls the disclosure of information and informs partners about the parts amenable to be personalized. Partners can adapt the content offered through company's application based on their customers preferences. This solution makes use of well-known programming paradigms, which facilitates the adoption by the company and their partners. The technical feasibility of this architecture is demonstrated using the conference website as an example.

- *Chapter* 4 allows scripters to personalize websites (*Hybrid Open Personalization*). There already exist scripting communities that personalize websites without the help of the website creators. In this chapter an architecture of participation is proposed, based on the usage of an interface for web modification. The website creator defines an interface that scripters use to create their scripts. Scripters benefits from this architecture by facilitating the creation of such modifications. Website creator benefits from this architecture by offering modifications made by scripters through its site. Web customers benefits from this architecture by permitting them to select the modifications made by their mates. This solution makes use of well-known programming paradigms, which facilitates the adoption by the company and scripters. To illustrate this architecture, an example based on a conference website is presented.

- *Chapter* 5 allows end users to personalize websites (*Client-Side Open Personalization*). We resort to *Domain Specific Languages* to empower end users to augment a website by adding content from other sites. Unlike the previous scenarios, now contribution is mainly thought for self-consumption and social sharing. No

collaboration of the websites is required. These ideas are borne out through the *Sticklet* language. Being an end-user tool, evaluation is conducted through a set of usability experiments.

6.3 Publications

Part of the work presented in this thesis has already been discussed and presented in different peer-reviewed forums. The author has contributed to the following publications:

Selected Publications

- Oscar Díaz, Cristóbal Arellano and Maider Azanza. A DSL for End-user Web Augmentation: Caring for Producers and Consumers Alike. Accepted ~~in~~ *ACM Transactions on the Web* [DAA]. JCR.

- Cristóbal Arellano, Oscar Díaz and Jon Iturrioz. Opening Personalization to Partners: An Architecture of Participation for Websites. In *12th International Conference on Web Engineering (ICWE 2012)*, Berlin, Germany, 2012 [ADI12]. Rank B in the *ERA Conference Ranking*. Acceptance rate: 20.4%.

- Cristóbal Arellano, Oscar Díaz and Jon Iturrioz. Crowdsourced Web Augmentation: A Security Model. In *11th International Conference on Web Information Systems Engineering (WISE 2010)*, Hong Kong, China, 2010 [ADI10a]. Rank A in the *ERA Conference Ranking*. Acceptance rate: 18.8%.

- Oscar Díaz, Cristóbal Arellano and Jon Iturrioz. Interfaces for Scripting: Making Greasemonkey Scripts Resilient to Website Upgrades. In *10th International Conference on Web Engineering (ICWE 2010)*, Vienna, Austria, 2010 [DAI10]. Rank B in the *ERA Conference Ranking*. Acceptance rate: 20.0%.

- Oscar Díaz, Cristóbal Arellano and Jon Iturrioz. Layman Tuning of Websites: Facing Change Resilience. In *17th International World Wide Web Conference (WWW 2008)*, Beijing, China, 2008 [DAI08]. Rank A in the *ERA Conference Ranking*.

International Conferences/Workshops

- Oscar Díaz, Cristobal Arellano, Gorka Puente. Wikipedia Customization through Web Augmentation Techniques. In *8th International Symposium on Wikis and Open Collaboration (WikiSym 2012)*, Linz, Austria, 2012. Rank B in the *ERA Conference Ranking*. Acceptance rate 55.0% [DAP12].

- Oscar Díaz and Cristóbal Arellano. Sticklet: An End-User Client-Side Augmentation-Based Mashup Tool. In *12th International Conference on Web Engineering (ICWE 2012)*, Berlin, Germany, 2012 [DA12].

- Oscar Díaz, Josune de Sosa, Cristóbal Arellano and Salvador Trujillo. Web-Based Tool Integration: A Web Augmentation Approach. In *12th International Conference on Web Engineering (ICWE 2012)*, Berlin, Germany, 2012 [DdSAT12].

- Oscar Díaz, Gorka Puente, Cristóbal Arellano. Wiki refactoring: an assisted approach based on ballots. In *7th International Symposium on Wikis and Open Collaboration (WikiSym 2011)*, Mountain View, USA, 2011. Rank B in the *ERA Conference Ranking*. Acceptance rate 42.0% [DPA11].

- Cristóbal Arellano, Oscar Díaz and Jon Iturrioz. Script Programmers as Value Co-creators. In *Enterprise Crowdsourcing Workshop held at 10th International Conference on Web Engineering (ICWE 2010)*, Vienna, Austria, 2010 [ADI10b].

- Oscar Díaz, Jon Iturrioz and Cristóbal Arellano. Facing Tagging Data Scattering. In *10th International Conference on Web Information Systems Engineering (WISE 2009)*, Poznan, Poland, 2009 [DIA09]. Rank A in the *ERA Conference Ranking*. Acceptance rate: 23.0%.

- Oscar Díaz, Sandy Pérez and Cristóbal Arellano. Tagging-Aware Portlets. In *9th International Conference on Web Engineering (ICWE 2009)*, San Sebastian, Spain, 2009 [DPA09]. Rank C in the *ERA Conference Ranking*. Acceptance rate: 24.0%.

- Cristóbal Arellano, Oscar Díaz and Jon Iturrioz. The Modding Web: Layman Tuning of Websites. In *9th International Conference on Web Engineering (ICWE 2009)*, San Sebastian, Spain, 2009 [ADI09].

- Jon Iturrioz, Oscar Díaz and Cristóbal Arellano. Towards Federated Web 2.0 Sites: The TAGMAS Approach. In *Tagging and Metadata for Social Information Organization Workshop held at 16th International World Wide Web Conference (WWW 2007)*, Banff, Canada, 2007 [IDA07].

6.4 Research Stage

A Ph.D. is a learning process in which the supervisor plays an important role. As a consequence, how the research problems are faced during this process is influenced by the supervisor. Trying to complement different perspectives of the research problems discussed in this dissertation, the author conducted a research stage. The author visited the *University of Utrecht, Netherlands*, under the supervision of *Dr. Slinger Jansen* from February to April of 2011. During this research, the author acquired a deepen knowledge in online communities which influenced the content of the Chapter 4.

6.5 Assessment and Future Research

In this dissertation, the author proposes three scenarios where the design of *Web Personalization* is opened to other actors. During the development of the solutions some limitations where detected. Such limitations mark the directions of future work.

Server-Side Open Personalization Implications

- *Extension of User Model by partners*: The partners' modifications are built on top of the User Model defined by the company's web application. This model might be insufficient to perform some mods. Mechanisms are needed to allow partners to request extra information to users. Additionally, partners could already have information in their web applications about the company's users. User profile interoperability is an open issue and the interest in the research community is reflected in the existence of workshops that are specialized in this topic such as *"International Workshop on Interoperability of User Profiles in Multi-Application Web Environments (MultiA-Pro 2012)*".

- *Business models*: Traditional business models, like the advertising model, need to be adapted to this new environment. ~~Main~~ company is benefited because the content of its website is improved. Additionally, ~~their~~ partners now have the chance to adapt their offers to a concrete user. This win-win relationship could be sufficient if the negotiation power of the company and its partners is similar. If the company has more negotiation power than the partners, it could introduce some rewards from their partners. Some business models need to be introduced to reflect these situations.

- *Permissions*: The partners have access to the parts of the *Domain Model* that appears in the *Modding Interface*. At this time, the *Modding Interface* is the same for all the partners and all the partners

have access to the all instances of the all *Modding Concepts*. Some mechanisms are needed to let the company specify the permissions over the *Modding Interface* depending on the partner. Additionally, the permissions need to be enforced, preventing the disclosure of information to unauthorized partners. Some work in this direction already exist [BSP12, ACD12].

Hybrid Open Personalization Implications

- *Promote contribution*: The construction of communities around web applications implies promoting/rewarding contributions, disseminating contributions through the web application, facilitating end users to suggest augmentations, and so on. In the same way that Web2.0 APIs open data silos to achieve application composition at the back-end, we envision *Modding Interfaces* “to open” application markup to crowdsourced, front-end composition.

- *Include augmentations in the core*: Some augmentations provided by users could be useful not only for a subset but for the whole website users. It is needed to explore the possibility of promoting these augmentations to be a part of the core of the hosting application.

Client-Side Open Personalization Implications

- *Access to desktop resources*: Although the tendency is to move resources to the cloud, the desktop still keeps an important set of confidential material. Since augmentation is a client technology, desktop resources can safely participate in the augmentation.

- *Web Augmentation in mobile devices*: Mobile users will benefit from the reduction in the number of interactions that augmentation brings. Users can augment their favourite websites to act as a hub to easily access companion sites without the need to type complex URLs or conduct lengthy searches. In a mobile environment, other

resources take more relevance and have to be taken into account for a successful mobile *Web Augmentation*, like the usage of data plans, battery or GPS.

- *Apply Programming by Demonstration techniques*: *Programming by demonstration (PbD)* is a technique that facilitates the creation of programs by end users. As stated in [Lie01], "in this approach (PbD), a software agent records the interactions between the user and a conventional direct-manipulation interface, and writes a program that corresponds to the user's actions". At this time, the creation of *sticklets* is facilitated with clone&own and *assisted* techniques. It is needed to explore how *programming by demonstration* can complement these techniques in order to improve the effectiveness and the satisfaction of the user.

- *Evaluation in the wild*: Further evaluation is required, mainly for the consumer aspects. This requires the existence of a real community of users.

6.6 Conclusions

Web Personalization has proven to be an adequate paradigm to improve the satisfaction of web customers. This paradigm takes webmasters as the designers of such personalizations. But it is impossible for webmasters to foresee, developed and maintain such a moving target. *Open Personalization* alleviates this situation by allowing third parties to make such adaptations.

This dissertation presents three scenarios of *Open Personalization.* The first scenario is *Server-Side Open Personalization* where partners are allowed to adapt company's website using software interfaces to regulate the adaptation. The second scenario is *Hybrid Open Personalization* where the adaptation of the website is made by scripters with the help of the webmaster. In the third scenario, *Client-Side Open Personalization*, end

users can personalize websites with no help from the webmaster but with the help of a *domain specific language*. Part of the content of this thesis has already been presented in different venues. To conclude, the author enumerated some limitations of the work which can serve as future lines of research.

Bibliography

[AA05] A9 and Amazon. OpenSearch 1.1 Spec. Online, 2005. http://www.opensearch.org/Specifications/OpenSearch/1.1 [accessed November 2012].

[ACD12] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Recovering Role-Based Access Control Security Models from Dynamic Web Applications. In *Proceedings of the 12th International Conference on Web Engineering, ICWE '12*, pages 121–136, 2012.

[ADI09] Cristóbal Arellano, Oscar Díaz, and Jon Iturrioz. The Modding Web: Layman Tuning of Websites. In *Proceedings of the 9th International Conference on Web Engineering, ICWE '09*, 2009.

[ADI10a] Cristóbal Arellano, Oscar Díaz, and Jon Iturrioz. Crowdsourced Web Augmentation: A Security Model. In *Proceedings of the 11th International Conference on Web Information Systems Engineering, WISE '10*, pages 294–307, 2010.

[ADI10b] Cristóbal Arellano, Oscar Díaz, and Jon Iturrioz. Script Programmers as Value Co-creators. In *Enterprise Crowdsourcing Workshop*, pages 417–420, 2010.

[ADI12] Cristóbal Arellano, Oscar Díaz, and Jon Iturrioz. Opening Personalization to Partners: An Architecture of Participation for Websites. In *Proceedings of the 12th International Conference on Web Engineering, ICWE '12*, pages 91–105, 2012.

[ÁPR+10] Manuel Álvarez, Alberto Pan, Juan Raposo, Fernando Bellas, and Fidel Cacheda. Finding and Extracting Data Records from Web Pages. *Signal Processing Systems*, 59:123–137, 2010.

[AVG10] AVG. AVG LinkScanner - How it Works. Online, 2010. http://linkscanner.avg.com/ww.sals-how-it-works.html [accessed November 2012].

[BBCS08] Christopher Bogart, Margaret M. Burnett, Allen Cypher, and Christopher Scaffidi. End-User Programming in the Wild: A Field Study of CoScripter Scripts. In *Proceedings of the 24th IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC '08*, pages 39–46, 2008.

[BBS10] Jan Bosch and Petra Bosch-Sijtsema. From Integration to Composition: On the Impact of Software Product Lines, Global Development and Ecosystems. *Journal of Systems and Software*, 83:67–76, 2010.

[BC98] Paul De Bra and Licia Calvi. 2l670: A Flexible Adaptive Hypertext Courseware System. In *Proceedings of the 9th ACM Conference on Hypertext and Hypermedia, HYPERTEXT '98*, pages 283–284, 1998.

[BE98] Peter Brusilovsky and John Eklund. A Study of User Model Based Link Annotation in Educational Hypermedia. *Universal Computer Science*, 4(4):429–448, 1998.

[BG99] Alan F. Blackwell and Thomas R. G. Green. Investment of Attention as an Analytic Approach to Cognitive Dimensions. In *Collected Papers of the 11th Annual Workshop of the Psychology of Programming Interest Group*, pages 246–253, 1999.

[BHP+06] Steffen Becker, Wilhelm Hasselbring, Alexandra Paul, Marko Boskovic, Heiko Koziolek, Jan Ploski, Abhishek Dhama, Henrik Lipskoch, Matthias Rohr, Daniel Winteler, Simon Giesecke, Roland Meyer, Mani Swaminathan, Jens Happe, Margarete Muhle, and Timo Warns. Trustworthy Software Systems: A Discussion of Basic Concepts and Terminology. *ACM SIGSOFT Software Engineering Notes*, 31:1–18, 2006.

[Bir05] Dorian Birsan. On Plug-ins and Extensible Architectures. *ACM Queue*, 3:40–46, 2005.

[BKGM+02] Orkut Buyukkokten, Oliver Kaljuvee, Hector Garcia-Molina, Andreas Paepcke, and Terry Winograd. Efficient Web Browsing on Handheld Devices Using Page and Form Summarization. *ACM Transactions on Information Systems*, 20:82–115, 2002.

[Bla02] Alan F. Blackwell. First Steps in Programming: A Rationale for Attention Investment Models. In *Proceedings of the 2002 IEEE Symposum on Human-Centric Computing Languages and Environments, HCC '02*, pages 2–10, 2002.

[BLK+09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A Crystallization Point for the Web of Data. *Journal Web Semantics: Science, Services and Agents on the World Wide Web*, 7:154–165, 2009.

[Bos09] Jan Bosch. From Software Product Lines to Software Ecosystems. In *Proceedings of the 13th International Software Product Lines Conference, SPLC '09*, pages 111–119, 2009.

[Bou99] Niels O. Bouvin. Unifying Strategies for Web augmentation. In *Proceedings of the 10th ACM Conference on Hypertext and Hypermedia, HYPERTEXT '99*, pages 91–100, 1999.

[Bru96] Peter Brusilovsky. Methods and Techniques of Adaptive Hypermedia. *User Modeling User-Adapted Interaction*, 6(2-3):87–129, 1996.

[Bru01] Peter Brusilovsky. Adaptive Hypermedia. *User Modeling and User-Adapted Interaction*, 11(1-2):87–110, 2001.

[BSP12] Mairon Belchior, Daniel Schwabe, and Fernando Silva Parreiras. Role-Based Access Control for Model-Driven Web Applications. In *Proceedings of the 12th International Conference on Web Engineering, ICWE '12*, pages 106–120, 2012.

[BSW96] Peter Brusilovsky, Elmar W. Schwarz, and Gerhard Weber. ELM-ART: An Intelligent Tutoring System on World Wide Web. In *Proceedings of the 3rd International Conference on Intelligent Tutoring Systems, ITS '96*, pages 261–269, 1996.

[BWR+05] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology, UIST '05*, pages 163–172, 2005.

[CCG11] Francesca Carmagnola, Federica Cena, and Cristina Gena. User Model Interoperability: A Survey. *User Modeling and User-Adapted Interaction*, 21:285–331, 2011.

[CDA00] Ibrahim Cingil, Asuman Dogac, and Ayca Azgin. A Broader Approach to Personalization. *Communications of the ACM*, 43(8):136–141, 2000.

[CDM+11] Cinzia Cappiello, Florian Daniel, Maristella Matera, Matteo Picozzi, and Michael Weiss. Enabling End User Development through Mashups: Requirements, Abstractions and Innovation Toolkits. In *Proceedings of the 3rd International Symposium - End-User Development, IS-EUD '11*, pages 9–24, 2011.

[Cro06] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). Technical report, Internet Engineering Task Force, 2006. http://tools.ietf.org/html/rfc4627 [accessed November 2012].

[CTB03] Sven Casteleyn, Olga De Troyer, and Saar Brockmans. Design Time Support for Adaptive Behavior in Web Sites. In *Proceedings of the 2003 ACM Symposium on Applied Computing, SAC '03*, pages 1222–1228, 2003.

[DA12] Oscar Díaz and Cristóbal Arellano. Sticklet: An End-User Client-Side Augmentation-Based Mashup Tool. In *Proceedings of the 12th International Conference on Web Engineering, ICWE '12*, pages 465–468, 2012.

[DAA] Oscar Díaz, Cristóbal Arellano, and Maider Azanza. A DSL for End-user Web Augmentation: Caring for Producers and Consumers Alike. *ACM Transactions on the Web*. Accepted.

[DAI08] Oscar Díaz, Cristóbal Arellano, and Jon Iturrioz. Layman Tuning of Websites: Facing Change Resilience. In *Proceedings of the 17th International Conference on World Wide Web, WWW '08*, pages 1127–1128, 2008.

[DAI10] Oscar Díaz, Cristóbal Arellano, and Jon Iturrioz. Interfaces for Scripting: Making Greasemonkey Scripts Resilient to Website Upgrades. In *Proceedings of the 10th International Conference on Web Engineering, ICWE '10*, pages 233–247, 2010.

[DAP12] Oscar Díaz, Cristóbal Arellano, and Gorka Puente. Wikipedia Customization through Web Augmentation Techniques. In *Proceedings of the 8th International Symposium on Wikis and Open Collaboration, WIKISYM '12*, 2012.

[Dav89] Fred D. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly*, 13:319–340, 1989.

[DCBS09] Florian Daniel, Fabio Casati, Boualem Benatallah, and Min-Chien Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In *Proceedings of the 28th International Conference on Conceptual Modeling, ER '09*, pages 428–443, 2009.

[DdSAT12] Oscar Díaz, Josune de Sosa, Cristóbal Arellano, and Salvador Trujillo. Web-Based Tool Integration: A Web Augmentation Approach. In *Proceedings of the 12th International Conference on Web Engineering, ICWE '12*, pages 431–434, 2012.

[DDSC07] Mira Dontcheva, Steven M. Drucker, David Salesin, and Michael F. Cohen. Changes in Webpage Structure over Time. Technical report, University of Washington, 2007.

[DIA09] Oscar Díaz, Jon Iturrioz, and Cristóbal Arellano. Facing tagging data scattering. In *Proceedings of the 10th International Conference on Web Information Systems Engineering, WISE '09*, pages 63–74, 2009.

[DLL06] Yvonne Dittrich, Olle Lindeberg, and Lars Lundberg. End-User Development as Adaptive Maintenance. In *End-User Development*, pages 295–313. Springer, 2006.

[DMP06] Florian Daniel, Maristella Matera, and Giuseppe Pozzi. Combining Conceptual Modeling and Active Rules for the Design of Adaptive Web Applications. In *Proceedings of the 1th International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE '06*, 2006.

[DPA09] Oscar Díaz, Sandy Pérez, and Cristóbal Arellano. Tagging-aware portlets. In *Proceedings of the 9th International Conference on Web Engineering, ICWE '09*, pages 61–75, 2009.

[DPA11] Oscar Díaz, Gorka Puente, and Cristóbal Arellano. Wiki Refactoring: An Assisted Approach Based on Ballots. In *Proceedings of the 7th International Symposium on Wikis and Open Collaboration, WIKISYM '11*, pages 195–196, 2011.

[DRC+12] Florian Daniel, Carlos Rodríguez, Soudip Roy Chowdhury, Hamid R. Motahari, and Fabio Casati. Discovery and Reuse of Composition Knowledge for Assisted Mashup Development. In *Proceedings of the 21st International*

World Wide Web Conference, WWW '12, pages 493–494, 2012.

[EBG+07] Rob Ennals, Eric A. Brewer, Minos N. Garofalakis, Michael Shadle, and Prashant Gandhi. Intel Mash Maker: Join the Web. *SIGMOD Record*, 36:27–33, 2007.

[EH98] Deborah M Edwards and Lynda Hardman. *Hypertext: Theory into Practice*, chapter 'Lost in hyperspace': cognitive mapping and navigation in a hypertext environment, pages 90–105. Intellect Ltd, 1998.

[eTe10] Eoin eTeanga. Create a custom search engine for Firefox, IE, Chrome. Online, 2010. http://www.eteanga.ie/create-a-custom-search-engine-for-firefox-ie-chrome/ [accessed November 2012].

[EW85] William C. Elm and David D. Woods. Getting Lost: A Case Study in Interface Design. In *Proceedings of the 29th Human Factors and Ergonomics Society Annual Meeting*, pages 927–929, 1985.

[Fac10a] Facebook. Facebook. Online, 2010. http://www.facebook.com/ [accessed November 2012].

[Fac10b] Facebook. Facebook Developers Platform. Online, 2010. http://developers.facebook.com/ [accessed November 2012].

[Fac10c] Facebook. Facebook Platform: How to use the new Facebook social plugins for your business. Online, 2010. http://www.facebook.com/notes/facebook-platform/how-to-use-the-new-facebook-social-plugins-for-your-business/394310302301 [accessed November 2012].

[FH02] Flavius Frasincar and Geert-Jan Houben. Hypermedia Presentation Adaptation on the Semantic Web. In *Proceedings of the 2nd International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems, AH '02*, pages 133–142, 2002.

[Fil06] Robert E. Filman. Taking Back the Web. *IEEE Internet Computing*, 10:3–5, 2006.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. Online, January 2004. http://martinfowler.com/articles/injection.html [accessed November 2012].

[Fow09] Martin Fowler. A Pedagogical Framework for Domain-Specific Languages. *IEEE Software*, 26:13–14, 2009.

[FR03] Ellen M. Friedman and Jerry L. Rosenberg. Web Load Testing Made Easy: Testing with WCAT and WAST for Windows Applications. In *Proceedings of the 29th International Conference Management Group Conference, CMG '03*, pages 57–82, 2003.

[GBC+06] Thomas R. G. Green, Ann E. Blandford, Luke Church, Chris R. Roast, and Steven Clarke. Cognitive Dimensions: Achievements, New Directions, and Open Questions. *Journal of Visual Languages & Computing*, 17:328–365, 2006.

[GGC03] Irene Garrigós, Jaime Gómez, and Cristina Cachero. Modelling Adaptive Web Applications. In *Proceedings of the IADIS International Conference WWW/Internet, ICWI '03*, pages 813–816, 2003.

[GGH10] Irene Garrigós, Jaime Gómez, and Geert-Jan Houben. Specification of Personalization in Web Application Design. *Information & Software Technology*, 52:991–1010, 2010.

[Gli89] E. Glinert. A In-Depth Look at Selected Visual Systems. In *Proceedings of Workshop on Visual Computing Environments, CHI '89*, 1989.

[Goo] Google. Google Caja: A Source-to-Source Translator for Securing Javascript-based Web Content. Online. http://code.google.com/p/google-caja/ [accessed November 2012].

[GPB91] Thomas R. G. Green, Marian Petre, and Rachel K. E. Bellamy. Comprehensibility of Visual and Textual Programs: A Test of 'Superlativism' Against the 'match-mismatch' Conjecture. In *Proceedings of the 4th Workshop on Empirical Studies of Programmers*, 1991.

[GPS11] Giuseppe Ghiani, Fabio Paternò, and Lucio D. Spano. Creating Mashups by Direct Manipulation of Existing Web Applications. In *Proceedings of the 3rd International Symposium - End-User Development, IS-EUD '11*, pages 42–52, 2011.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004.

[HN99] Nicola Henze and Wolfgang Nejdl. Adaptivity in the KBS Hyperbook System. In *Proceedings of the 2th Workshop on Adaptive Systems and User Modeling of the WWW*, 1999.

[How06] Jeff Howe. The Rise of Crowdsourcing. Wired Magazine, 2006.

http://www.wired.com/wired/archive/14.06/crowds.html [accessed November 2012].

[HT10] Phillip Heidegger and Peter Thiemann. Contract-Driven Testing of JavaScript Code. In *Proceedings of the 4th International Conference on Tests & Proofs, TAP '10*, 2010.

[HV09] Stephan Hagemann and Gottfried Vossen. ActiveTags: Making Tags More Useful Anywhere on the Web. In *Proceedings of the 20th Australasian Database Conference, ADC '09*, pages 41–48, 2009.

[IDA07] Jon Iturrioz, Oscar Díaz, and Cristóbal Arellano. Towards Federated Web 2.0 Sites: The TAGMAS Approach. In *Tagging and Metadata for Social Information Organization Workshop*, 2007.

[IL10] Thierry Isckia and Denis Lescop. Open Innovation within Business Ecosystems: A Tale from Amazon.com. *Communications & Strategies*, 74:37–54, 2010.

[ISO01] ISO/IEC. Software Engineering - Software Product Quality - Part 1: Quality Model, 2001.

[JCP03] JCP. JSR 168: Portlet Specification Version 1.0. Online, 2003. http://www.jcp.org/en/jsr/detail?id=168 [accessed November 2012].

[JGM07] Afraz Jaffri, Hugh Glaser, and Ian Millard. URI Identity Management for Semantic Web Data Integration and Linkage. In *Proceedings of the 3rd International Workshop On Scalable Semantic Web Knowledge Base Systems*, pages 1125–1134, 2007.

[JWM10] Markus Jahn, Reinhard Wolfinger, and Hanspeter Mössenböck. Extending Web Applications with Client and

Server Plug-ins. In *Software Engineering*, pages 33–44, 2010.

[KAB+11] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. The State of the Art in End-User Software Engineering. *ACM Computing Surveys*, 43:21:1–21:44, 2011.

[Kap06] Mike Kaply. Operator Add-on for Firefox. Online, 2006. https://addons.mozilla.org/firefox/addon/operator/ [accessed November 2012].

[KC09] Rick Kazman and Hong-Mei Chen. The Metropolis Model: A New Logic for Development of Crowsourced Systems. *Communications of the ACM*, 52:76–84, 2009.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.

[KKP01] Alfred Kobsa, Jürgen Koenemann, and Wolfgang Pohl. Personalised Hypermedia Presentation Techniques for Improving Online Customer Relationships. *The Knowledge Engineering Review*, 16(2):111–155, 2001.

[Koc01] Nora Koch. *Software Engineering for Adaptive Hypermedia Systems*. PhD thesis, Ludwig-Maximilians-Universität München, 2001.

[KP88] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm

in Smalltalk-80. *Journal of Object Oriented Programming*, 1:26–49, 1988.

[KPW06] Markus Klann, Fabio Paternò, and Volker Wulf. Future Perspectives in End-User Development. In *End User Development*, pages 475–486. Springer, 2006.

[LBS05] Anthony Lieuallen, Aaron Boodman, and Johan Sundström. Greasemonkey. Online, 2005. http://www.greasespot.net/ [accessed November 2012].

[LE07] Sandeep Lingam and Sebastian G. Elbaum. Supporting End-Users in the Creation of Dependable Web Clips. In *Proceedings of the 16th International Conference on World Wide Web, WWW '07*, pages 953–962, 2007.

[LHML08] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. CoScripter: Automating & Sharing How-To Knowledge in the Enterprise. In *Proceedings of the 26th International Conference on Human Factors in Computing Systems,CHI '08*, pages 1719–1728, 2008.

[Lie01] Henry Lieberman, editor. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[MBH07] Knud Möller, Sean Bechhofer, and Tom Heath. Semantic Web Conference Ontology. Online, 2007. http://data.semanticweb.org/ns/swc/ontology [accessed November 2012].

[McC07] Adam McCrea. Metaprogramming JavaScript. Online, 2007. http://www.scribd.com/doc/522145/metaprogramming-javascript [accessed November 2012].

[McF05] Nigel McFarlane. Fixing Web Sites with Greasemonkey. *Linux Journal*, 138:1, 2005.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37:316–344, 2005.

[Mica] Microsoft. Internet Explorer 9 Accelerators. Online. http://windows.microsoft.com/en-US/windows7/How-to-use-Accelerators-in-Internet-Explorer-9 [accessed November 2012].

[Micb] Microsoft. Microsoft Web Sandbox: QoS Layer. Online. http://websandbox.livelabs.com/documentation/vm_qos.aspx [accessed November 2012].

[MM00] Robert C. Miller and Brad A. Myers. Integrating a Command Shell Into a Web Browser. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 171–182, 2000.

[MP09] John J. Maver and Cappy Popp. *Essential Facebook Development: Build Successful Applications for the Facebook Platform*. Addison-Wesley, 2009.

[MS95] Salvatore T. March and Gerald F. Smith. Design and Natural Science Research on Information Technology. *Decision Support Systems*, 15(4):251–266, 1995.

[Mul12] International Workshop on Interoperability of User Profiles in Multi-Application Web Environments (MultiA-Pro 2012). Online, 2012. http://liris.cnrs.fr/multiapro2012/ [accessed November 2012].

[Mye90] Brad A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[NT08] Sagar Naik and Piyu Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2008.

[O'R04] Tim O'Reilly. The Architecture of Participation. Online, June 2004. http://oreilly.com/pub/a/oreilly/tim/articles /architecture_of_participation.html [accessed November 2012].

[Par72] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15:1053–1058, 1972.

[PD99] Norman W. Paton and Oscar Díaz. Active Database Systems. *ACM Computing Surveys*, 31:63–103, 1999.

[PD10] Iñaki Paz and Oscar Díaz. Providing Resilient XPaths for External Adaptation Engines. In *Proceedings of the 21st ACM Conference on Hypertext and Hypermedia, HYPERTEXT '10*, pages 67–76, 2010.

[PGKM03] Kyparisia A. Papanikolaou, Maria Grigoriadou, Harry Kornilakis, and George D. Magoulas. Personalizing the Interaction in a Web-based Educational Hypermedia System: the case of INSPIRE. *User Modeling and User-Adapted Interaction*, 13(3):21–267, 2003.

[RBG12] Raphael M. Reischuk, Michael Backes, and Johannes Gehrke. SAFE Extensibility for Data-Driven Web Applications. In *Proceedings of the 21st International World Wide Web Conference, WWW '12*, pages 799–808, 2012.

[RI06] Alexander Repenning and Andri Ioannidou. What Makes End-User Development Tick? 13 Design Guidelines. In *End User Development*, pages 51–85. Springer, 2006.

[Rie11] Mikko Riepula. Sharing Source Code with Clients: A Hybrid Business and Development Model. *IEEE Software*, 28(4):36–41, 2011.

[RSG01] Gustavo Rossi, Daniel Schwabe, and Robson Guimarães. Designing Personalized Web Applications. In *Proceedings of the 10th International World Wide Web Conference, WWW '01*, pages 275–284, 2001.

[SBVG10] David Sánchez, Montserrat Batet, Aida Valls, and Karina Gibert. Ontology-driven Web-based Semantic Similarity. *Journal of Intelligent Information Systems*, 3:383–413, 2010.

[Sch01] Jörg Schaible. JsUnit. Online, 2001. http://jsunit.berlios.de/ [accessed November 2012].

[Sky05] Skype. Skype button in Internet Explorer or Firefox toolbar. Online, 2005. http://www.skype.com/intl/en/support/user-guides/toolbar?lang=en [accessed November 2012].

[SWM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL Web Ontology Language Guide. Online, 2004. http://www.w3.org/TR/owl-guide/ [accessed November 2012].

[TDD+09] Michael Toomim, Steven M. Drucker, Mira Dontcheva, Ali Rahimi, Blake Thomson, and James A. Landay. Attaching UI Enhancements to Websites with End Users. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI '09*, pages 1859–1868, 2009.

[The11] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4.3, 2011.

[TKS11] Rattapoom Tuchinda, Craig A. Knoblock, and Pedro A. Szekely. Building Mashups by Demonstration. *ACM Transactions on the Web*, 5:16:1–16:45, 2011.

[Tur05] Scott R. Turner. Platypus. Online, 2005. http://platypus.mozdev.org/ [accessed November 2012].

[UMA12] The 20th Conference on User modeling, Adaptation, and Personalization (UMAP 2012). Online, 2012. http://umap2012.polymtl.ca/en/ [accessed November 2012].

[W3C00a] W3CDOMWG. Document Object Model (DOM) Level 2. Online, 2000. http://www.w3.org/DOM/DOMTR#dom2 [accessed November 2012].

[W3C00b] W3CDOMWG. Document Object Model (DOM) Level 2 Core Specification. Online, 2000. http://www.w3.org/TR/DOM-Level-2-Core [accessed November 2012].

[W3C08] W3CHTML5WG. HTML5. Online, 2008. http://www.w3.org/TR/html5/ [accessed November 2012].

[Wik10a] Wikipedia. Design by Contract at Wikipedia. Online, 2010. http://en.wikipedia.org/wiki/Design_by_contract [accessed November 2012].

[Wik10b] Wikipedia. Monkey Patch at Wikipedia. Online, 2010. http://en.wikipedia.org/wiki/Monkey_patch [accessed November 2012].

[Win06] William E. Winkler. Overview of Record Linkage and Current Research Directions. Technical report, U.S. Bureau of the Census, 2006.

[WS97] Gerhard Weber and Marcus Specht. User Modeling and Adaptive Navigation Support in WWW-Based Tutoring Systems. In *Proceedings of the 6th International Conference on User Modeling, UM '97*, pages 289–300, 1997.

[XLH$^+$09] Xiangye Xiao, Qiong Luo, Dan Hong, Hongbo Fu, Xing Xie, and Wei-Ying Ma. Browsing on Small Displays by Transforming Web Pages into Hierarchically Structured Subpages. *ACM Transactions on the Web*, 3:4:1–4:36, 2009.

[Yah07] Yahoo. Yahoo Pipes. Online, 2007. http://pipes.yahoo.com/pipes/ [accessed November 2012].

[YBCD08] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding Mashup Development. *IEEE Internet Computing*, 12:44–52, 2008.

[YLZ09] Xingliang Yu, Jing Li, and Hua Zhong. On Reducing the Pre-release Failures of Web Plug-in on Social Networking Site. In *Proceedings of the 2009 International Conference on Software Process, ICSP '09*, pages 236–245, 2009.

Acknowledgments

First of all, I wish to express my most sincere gratitude to my supervisors, Prof. Dr. Oscar Díaz and Dr. Jon Iturrioz. They have not only advised me during the whole Ph.D. but we have been working together as a team, combining the strengths of each. From the personal viewpoint, they treated me superb.

Whereas my supervisors give me assistance in terms of guidance, ONEKIN, the hosting research group assisted me in other directions. Thanks to Sandy Pérez for the web discussions and LaTeX tricks, to Gorka Puente that put me in touch with the hottest technological news and the innumerable assistance during the writing of the dissertation and to Maider Azanza that helped me to do things in the right way and her dessert recipes. To Josune De Sosa, Jokin Pérez, Iñigo Aldalur and Itziar Otaduy that have had infinite patience and have participated in the laboratory talks. To Arantza Irastorza, Iker Azpeitia, Felipe Ibañez, Felipe Martín and Iñaki Paz that have helped me directly or indirectly; the research group follows the communicating vessels theory and they helped me sometimes without I have noticed it. To Salvador Trujillo that gave me some invaluable advises.

This thesis was economically supported by the Spanish Ministry of Education and Science through the under the FPI Program. It has allowed me to be independent from the economical point of view and has supported the research stage at Utrecht (Netherlands).

To my colleagues at Utrecht University, Amir Saedi, Ravi Kadhka and Michiel Meulendijk that made the research stage easier. To Slinger Jansen, that welcomed and integrated me in his research group.

Thanks to my family and concretely to my parents that let me to choose my way and put their resources that were essentials to reach to this point. To my friends, that unconditionally they have been there and have understood my occasional absences.

Last but not least, I want to give my best thanks to Aiora; we have gone hand in hand and she helps me to grow not only at the professional dimension but the personal one.