Towards Context-Aware Web Applications

Po-Hao Chang and Gul Agha

University of Illinois at Urbana-Champaign,
201 North Goodwin Avenue, Urbana IL 61801, USA
{pchang2, agha}@cs.uiuc.edu

Abstract. In order to guarantee certain levels of QoS, a Web application needs to adapt itself to different execution contexts. However, because of the lack of coordination support in Web platforms, service providers respond to the challenge by simply providing multiple versions of a Web application, one for each context. We argue this top-down approach is neither efficient nor scalable: developing a context-specific application requires considerable effort and expertise while the ever-changing Internet never stops generating interesting contexts which can be exploited for better deployment. As an alternative, we propose a three-layer, bottom-up approach to building context-aware Web applications. At the bottom layer, we characterize a context-specific Web application with a particular *component distribution plan* which provides details for composing individual objects. In the middle layer, recursively defined *configurations* provide a bridge which relates high-level context features to low-level component distribution properties, where a configuration is a combination of configurations and/or component distribution properties. At the top level, a *context management system* selects desirable configurations according to the execution contexts.

1 Introduction

Evolving from its original mission of content delivery, the Web has become a gateway of assorted interactive applications: people access emails, shop online, trade equities, manage accounts and even remotely control home appliances using various Web applications. Unlike its typical standalone and distributed peers, a Web application encounters heterogeneous execution environments and numerous, unpredictable circumstances in its deployment. From the early days of the Web, developers identified the need to differentiate execution contexts: it was common for a Web application to have one version with *HTML frames* and another without. Context differentiation has become more critical as the Web has evolved: new crop of context-specific versions such as *broadband*, *JavaScript-enabled*, *HTML only* and *low graphics* can be found in many Web applications.

In most cases, these versions are built in a top-down fashion: that is, given a context, the programmer exploits its features and develops a specific version accordingly. However, the development and maintenance cost of multiple versions is always expensive, and in the case of Web applications, this approach is fragile in two ways:

1. Web applications are usually evolutionary in their life cycles. A minor change in the application requirement may result in a re-evaluation of features to be exploited in the contexts of interest.
2. The domain of interesting contexts is changing. Some contexts have fallen into disuse and others are gaining sufficient momentum to require particular attention. Since the future trend is hardly predictable, not much of existing code can be readily reused.

We believe a bottom-up approach is a more feasible way to build and manage a context-aware Web application. This is motivated by the following observation: no matter how many versions the application supports, there are some elements in common, such as its core application logic. We model a Web application as a composition of distributed objects: the composing structure and the functions of individual objects characterize the application. Instead of building a monolithic context-specific version from scratch, the developer picks and/or adapts composing objects with desired attributes, for example, implementation technology, execution location and deployment policy, to fit for the context of interest. The idea is similar to Web styling sheets [7, 13]: a customized presentation of a document can be achieved by supplying a specific style sheet.

The ability to customize a Web application through object annotations is just on the halfway to context-aware Web applications: the customizable application requires human intervention (meta-programming) to adapt into contexts. To be truly context-aware, the application needs assistance from a context management system to automate this process. The goal of the system is to generate detailed deployment plans based on the context features at runtime. However, using straight-forward reasoning from context features to desired object attributes complicates policy-design and suffers similar difficulties to those described above. We use modularity to address the problem: a full deployment plan is decomposed into sub-plans, each of which determines a set of closely-related object annotations, and a policy associates a context feature to one or more sub-plans only. A full deployment plan can be decided as the context management system applies the policies applicable to an incoming context.

In this paper, we describe a software system to support context-aware Web applications. Our system follows the bottom-up approach and enables a Web application to adapt its component distribution to different execution contexts. The rest of this paper is structured as follows: Section 2 provides a comprehensive background of the problem and an overview of our strategies. Section 3 introduces a component framework supporting customizable Web applications through annotating composing objects. In section 4 we present a structural and parameterized representation of related annotations akin to potential context features. Section 5 describes a context management system which follows user-defined policies to generate context-specific deployment plans from applicable object annotations – and thus makes Web applications context-aware. We conclude the paper with a discussion in the final section.

2 Overview

Web applications are inherently distributed: they require cooperation between a server and a client in order to accomplish their tasks. We argue that component distribution, and particularly execution location and loading policy, has a strong impact on the performance of Web applications in different contexts. Therefore, we propose enabling a Web application to adapt its distribution to the execution context. We first motivate the problem, discuss related work, and outline design strategies.

2.1 The Need for Context-Aware Web Applications

The goal of Web applications is to be accessible regardless of the platform a client is executing on. However, limitations in the capability of a client restricts what object distributions it can support: for example, a thin client cannot perform complex computing tasks and has limited control over application loading. The limitations posed by a thin client simplify the problem of object distribution: there are no choices to be made in determining the component distribution. However, as more and more clients support *AJAX* [8] Web applications–which require a full-fledged computing platform in the client–component distribution becomes an important factor in ensuring certain levels of QoS. The examples below illustrate how an execution context may favor certain distribution plans.

Location: For computing components which require no input from the server and consume few CPU cycles, such as a unit converter or a mortgage calculator, it is preferable to deploy them in the client both for a faster response time and to create less workload in the server. In other cases, the best location is not always clear: a CPU-intensive component which takes input from a backend data source is usually better allocated in the server because JavaScript is not an efficient way to do the computation and bringing data across the Internet is a significant overhead; however, in cases where the server CPU is extremely busy but the server is less stressed in I/O and bandwidth, it is better to shift the component to the client. A case of this sort that we have seen in practice is a component which extracted excerpts of documents based on a user's query. The computation consumes many more CPU cycles than searching and fetching the documents, although it still finishes in milliseconds when the server is lightly loaded. When the server is busy doing multiple tasks (e.g., processing other requests or indexing documents), it can take dozens of seconds to return the excerpts; in such a context, it is more efficient to deploy the component in the client.

Timing: Many user interface controls, such as layered menus, list boxes and detailed information panels, have multiple levels of presentation. These components can be *preloaded* as their containers load, or loaded *on demand* when a user's action explicitly requires it. Preloading client components provides better response time but wastes bandwidth: some of these components are never used. We have investigated the effect on a TV listing application; preloading all the detailed information consumed double the bandwidth compared to loading on demand. A smarter solution is to exploit the user's profile: if certain preference can be identified, the application preloads only frequently used components. In this case, the preferable distribution depends on the current network utilization and identifiable usage patterns.

2.2 Problem Analysis and Related Work

It is desirable to make Web applications context-aware. Specifically, these applications require:

- The ability to *adapt* themselves to specific contexts of their deployment.
- The potential to *evolve* under widespread change in both execution environment and patterns of usage.

There are quite a few systems supporting context-aware applications under specific assumptions. Although their design concepts and principles can be applied to Web applications, there are several difficulties in using these systems in the domain of Web applications. We describe the difficulties below.

A key element of context-aware applications is adaptability. The execution environment of Web applications is heterogeneous: clients and servers usually employ incompatible technologies and assume different roles, which complicates the process of adaptation. Several research projects [2, 9, 14–17, 19] have been able to support location-transparent application development in distributed platforms; some of them are targeted to Web applications. One limitation in these systems is that the adaptability is restricted to component execution location: component distribution timing, which is crucial in many Web applications, is missing.

Another common limitation is in the mechanism to express and enforce deployment plans. Some systems [2, 15, 19] require *metaprograms* [11] (separate programs which manipulate programs) to control the distribution at runtime. This approach is not feasible in Web platforms because of the lack of rich runtime support. In [9, 14] the adaptivity is embedded in the library design: developers have to provide and use different libraries to reconfigure applications. *XML11* [16, 17] supports customization through separate specifications because the components are truly portable in various platforms natively; however, it is not clear yet how to construct specifications systematically.

Conceptually, the deployment plans contain information about component distribution *aspects*–concerns that are orthogonal to the application logic–and thus principles akin to *Aspect-Oriented Programming* (AOP) [1,5] can be applied as in [10, 18]. We observe that the complexity in aspect design has hindered its acceptance in Web applications: to facilitate fast prototyping and frequent modification, most Web applications are written in a less constrained fashion using scripting languages. In addition, the use of aspect programming results in an over specification of the requirements in a deployment plan and complicates the design of the context management system.

Context-aware software and service adaptation have been extensively studied and have gained success in pervasive computing [12] and multimedia QoS [6] adaptation; however, there are several assumptions in the case of adaptive component distribution in Web applications:

- It is acceptable to have a few bad deployments since there are several factors in the Internet which cannot be observed and predicted, such as actual network condition and client's stability. It is more important to ensure overall efficiency instead of optimal allocation in each execution.
- Resource consumption in a single execution is usually not demanding and the service duration is comparatively short. The pressure on the server system comes from numerous concurrent sessions, not individual sessions. This has two implications:
  - Complex decision-making processes such as negotiation may kill any benefit gained through adaptation.
  - The ability to re-adapt (under context change) during a session is not crucial.
- The Web is an *open system* composed of standards and protocols. A solution requiring extra features in all participating platforms is unlikely to win wide acceptance.

2.3 Design Strategies

From the analysis above, we identify two requirements of context-aware Web applications: *adaptability* and *extensibility.* We follow the principle of *separation of concerns* [4] in design to ensure adaptability, and adopt the paradigm of *generative programming* in implementation to guarantee extensibility. In the bottom layer, component distribution is separated from application logic and thus can be reconfigured according to separate specifications. A generative framework allows new distribution features to be added in the future. In the middle layer, features related to higher-level concepts are abstracted from component distribution rules and new features can be exploited using new transformation processes. In the top layer, context features are rendered into context variables which are used in defining deployment policies. New context features can be imported through new context variables with modules to collect them.

3 Customizable Web Applications

We have designed and implemented a component framework (Figure 1) to support reconfigurable component distribution. The basic idea is to separate component design and distribution features. In our framework, a *prototype* represents a design concept of component; the implementation of a component is synthesized by a generator with the distribution features that have been specified separately. The implementation details and algorithms used for synthesis can be found in [3]. In this section, we describe the extensible specification system which enables customizable Web application through distribution reconfiguration.

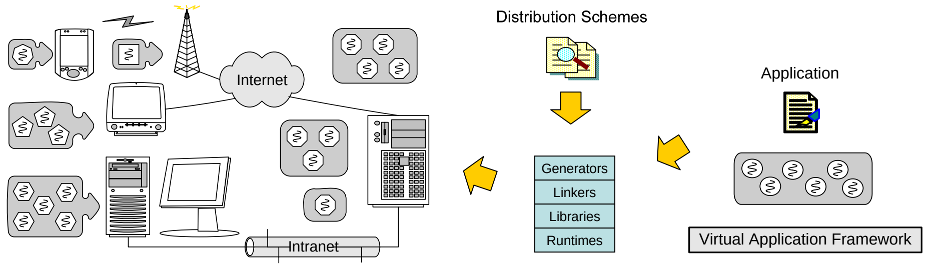

Fig. 1. Distribution transparent component framework.

3.1 Component Annotation

Many non-functional concerns, including those we are particularly interested in, can be specified by annotating components. For example, in order to specify the execution location, an attribute **Location** can be defined. However, it is not generally possible to annotate a specific component without knowing its unique identification. Instead of annotating specific individual components, we apply a rule to the set of all components that are created by a prototype. This turns out to be reasonable in the applications that we have looked at. The syntax for specifying that all components created with prototype X have the same *value* of **attribute A** is given below, together with an example of its use:

```
[prototype X]:[attribute A] = value;

DateValidator:Location = Client;
```

Rules of this form are suitable for large or unique components, but not for small ones which are used for different purposes. For example, `Button` is a common component; however, we expect that buttons have different attribute values in different circumstances.

3.2 Selection by Genealogy

An obvious candidate to further distinguish a component is its creator. We can select a set of components not only by their prototypes, but also by their creators' prototypes. This motivates the second rule for our specification scheme. The syntax to specify that all Y's created by an X share the same *value* of **attribute A** and examples of its use are shown below:

```
[prototype X]>[prototype Y]:[attribute A] = value;

OrderForm>Button:Location = Client;
InventoryForm>Button:Location = Server;
```

A generalization of the creator relation is to specify a component by its *genealogy*–extending creatorship to more generations. Note that the genealogy can be determined at creation time and remains invariant for a component. For example, the following rule says the attribute value of a component of prototype X_i is decided by examining its genealogy for up to i generations.

```
[X_0]>...>[X_i]: [attribute A] = value;
```

Obviously, examining the rules for more than one generation can lead to conflicting rules for the same attributes. We use the principle that a more specific rule overrides a less specific one. Because the genealogical ordering is linear, this serves to resolve conflicts.

3.3 Discussion

Annotating a set of components of a prototype with specific attributes is useful if the components use a specialized implementation of that prototype which produces components obeying the given specification. Note in our model, a prototype is a "concept of design" instead of an "implementation of design." From another prospective, a specification rule annotates a prototype implementation. Currently we define an attribute controlling the loading policy of prototype **PrototypeLoad** with two possible values *PreLoad* and *OnDemand*:

```
SubMenu:PrototypeLoad = OnDemand;
PricePanel>GridControl:PrototypeLoad = PreLoad;
CalcPanel>GridControl:PrototypeLoad = OnDemand;
```

The current implementation supports the following attributes: **Load** to control the component's loading policy, **PrototypeLoad** to the prototype's loading policy and **Location** to the component's execution location. Although only three attributes are supported, interesting attributes can be defined using supporting generators. For example, to support component mobility at runtime, we can add an attribute value `mobile` to **Location** and implement a generator synthesizing mobile components.

4 Structured Deployment Plans

It turns out that using the specification rules described directly is verbose and error-prone for human developers. Two attribute annotations may be combined in a rule if the target genealogy is the same, but two genealogies must be written in two rules even when they differ in only one generation. In addition, not all attributes are available for a prototype and some attribute values are in conflict with others. For example, the attribute **PrototypeLoad** is only applicable to a client implementation: a component cannot have this attribute with *Server* in **Location**. It is also difficult to manage and reuse individual specification rules. In a specification scheme containing a large number of complex rules, there is a greater chance that it has common building blocks that are reusable.

```
<OrderForm Location="Client">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <TaxCalculator Location="Client"/>
    <AddressValidator Location="Server"/>
</OrderForm>

OrderForm : Location =  Client;
OrderForm > ListControl : Location = Client;
OrderForm > ListControl > ListItem : Location = Client;
OrderForm > ListControl > ListItem : PrototypeLoad = OnDemand;
OrderForm > TaxCalculator : Location = Client;
OrderForm > AddressValidator : Location = Server;
```

Fig. 2. Using XML to represent specification rules.

4.1 Moving to XML

We use XML to organize specification rules: an XML element represents a prototype and multiple rules can be expressed in a tree structure. For example, the XML fragment and rules in Figure 2 are equivalent:

Using XML helps the developer to structure specification rules. Although it is legal to have a rule of a genealogy starting with a sub-component such as `SubMenuItem`, this makes little sense in practice. Instead, a set of specifications usually starts from a major component, such as `OrderForm` in our example. Another advantage of using XML is the existence of XML schema validation tools which can check validity and consistency of our specification rules. Note that adopting XML does not sacrifice expressiveness: any specification rule can be expressed in one XML fragment where every node has at most one child.

4.2 Parameterized Specification Blocks

Consider a specification scheme in the first part of Figure 3. The specifications on `OrderForm` and `ProfileForm` have a common building block highlighted in the grey areas. The observation immediately leads us to a shorthand representation in the second part of Figure 3. The specification scheme defines an XML `Block` element containing the common block with an attribute *name*, which can be used to refer the whole block in other specifications. The idea behind this is to make use of XML's tree structure: a node can readily refer to a set of subtrees with a modular representation.

```
<OrderForm Location="Client">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <TaxCalculator Location="Client"/>
    <AddressValidator Location="Server"/>
</OrderForm>

<ProfileForm Location="Client">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <PhoneValidator Location="Client"/>
    <AddressValidator Location="Server"/>
</ProfileForm>
```

```
<Block name="block1">
    <ListControl Location="Client">
        <ListItem Location="Client" PrototypeLoad="OnDemand"/>
    </ListControl>
    <AddressValidator Location="Server"/>
</Block>

<OrderForm Location="Client">
    <TaxCalculator Location="Client"/>
    <block1/>
</OrderForm>

<ProfileForm Location="Client">
    <PhoneValidator Location="Client"/>
    <block1/>
</ProfileForm>
```

Fig. 3. A common block can be defined by a `Block` element.

```
<Block name="block1">
      <AddressValidator Location="Server"/>
      <Configuration name="conf1">
             <ListControl Location="Client">
                   <ListItem Location="Client" PrototypeLoad="OnDemand"/>
             </ListControl>
      </Configuration>
      <Configuration name="conf2">
             <ListControl Location="Client">
                   <ListItem Location="Client" PrototypeLoad="PreLoad"/>
             </ListControl>
      </Configuration>
</Block1>

<OrderForm Location="Client">
      <TaxCalculator Location="Client"/>
      <block1 configuration="conf1"/>
</OrderForm>

<ProfileForm Location="Client">
      <PhoneValidator Location="Client"/>
      <block1 configuration="conf1"/>
</ProfileForm>

<FriendsList Location="Client">
      <EmailValidator Location="Client"/>
      <block1 configuration="conf2"/>
</FriendsList>
```

Fig. 4. A block can define multiple configurations.

However, using an element to represent a fixed set of subtrees is not as useful as it seems to be. If there is no other rule in Figure 3, it is not necessary to define a `Block` for `ListControl` and `AddressValidator` because `OrderForm` and `ProfileForm` have the same specification on these prototypes; top-level specifications on `ListControl` and `AddressValidator` are sufficient: `OrderForm` and `ProfileForm` will follow. A reusable block must be parameterized: it does not represent a set of rules (with fixed attribute values), but a group of *selectors*; the actual attribute values of these selectors, *configuration of the block*, can be controlled through a parameter.

We introduce another tag `Configuration` for configurations in a `Block`. Each `Configuration` element in a `Block` has the *name* attribute and contains an XML fragment representing the configuration. To reuse a block, we can set the *configuration* attribute to choose the desired configuration in the block. Figure 5 shows the expanded specification from Figure 4. Blocks and configurations can be constructed recursively: a `Configuration` element can contain other blocks. In addition, a `Block` element can contain elements other than `Configuration`: these elements will be included in the block replacement no matter which configuration is selected. (See `AddressValidator` specification in `block1`.)

```
<OrderForm Location="Client">
      <TaxCalculator Location="Client"/>
      <AddressValidator Location="Server"/>
      <ListControl Location="Client">
            <ListItem Location="Client" PrototypeLoad="OnDemand"/>
      </ListControl>
</OrderForm>

<ProfileForm Location="Client">
      <PhoneValidator Location="Client"/>
      <AddressValidator Location="Server"/>
      <ListControl Location="Client">
            <ListItem Location="Client" PrototypeLoad="OnDemand"/>
      </ListControl>
</ProfileForm>

<FriendsList Location="Client">
      <EmailValidator Location="Client"/>
      <AddressValidator Location="Server"/>
      <ListControl Location="Client">
            <ListItem Location="Client" PrototypeLoad="PreLoad"/>
      </ListControl>
</FriendsList>
```

Fig. 5. The expanded specification of Figure 4.

4.3 Partial Plans

XML is sufficiently expressive to represent an application's composition structure; XML can also organize cross-cutting concerns: logically unrelated components sharing common properties can be aggregated into a specification block. For example, a developer can identify those objects whose deployment have great impact on a certain resource (hence share a common property), such as CPU cycles and bandwidth, and define a specification block accordingly. The block then serves as a *partial plan* on condition of the specific resource. The context management system can reuse partial plans to create a deployment plan for a new identified context preference. This approach also provides extensibility: as new resources are taken into consideration, new specification blocks and new configurations can be designed independently without drastic changes in existing ones.

5 Context Management

We have designed and implemented an extensible context management system (Figure 6). The system includes three modules: *context monitors* active collect context information and store context features in *context variables*, which are used by *Adaptation Policies* to generate full deployment plans.

5.1 Context Features

The concept of context is abstract and the available features of a context are evolving. For example in the past service providers had little access to informa-

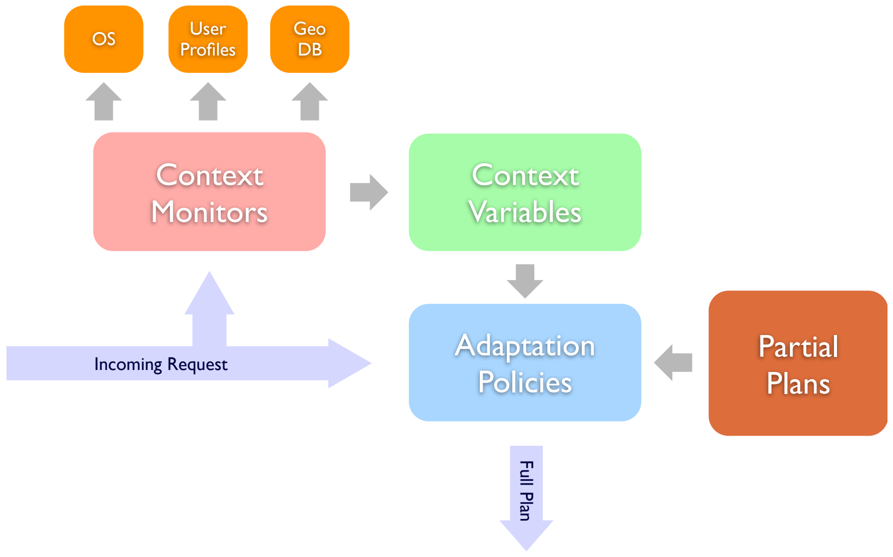

Fig. 6. The context management system.

tion about the client's *geolocation*, which is widely exploited nowadays for better service and resource allocation. Nonetheless, a context feature can be utilized only if it is quantitative and measurable. In our context management system, a context feature is represented by a *context variable*, and the introduction of a new context variable must come with a variable monitor maintaining the value.

Monitors can be implemented in a variety of forms as long as they update their variables in a timely manner. For example, the system status monitor for `System.CPU` and `System.Bandwidth` is implemented with OS system calls; the monitor for client capabilities is implemented with JavaScript detection code; and the user preference monitor reports related variable values by consulting the user profile database.

5.2 Policy Design

Defining an adaptation policy is straightforward: a policy is pair of a condition on context variables and a set of partial plans. When the context management system receives an incoming context, it collects the partial plans in the policies whose conditions are evaluated true and generates the full deployment plan.

Consider the first example in Section 2: we want to deploy the *Highlighter* to the client only under a special context where the CPU cycles are much more precious than the bandwidth, the policy can be written as follows:

(System.CPU*[$\texttt{Cost}_{cpu}$] > System.Bandwidth*[$\texttt{Cost}_{bandwidth}$])
=> <Highlighter Location="Client"/>

A cost function can be a constant for the normalization factor or a function of other context variables. In the second example, first we define a specification block **TVListing** with a configuration *FastResponse* which specifies all client components as *PreLoad*. The following policy selects the partial plan for fast response when the client's connection has long latency and the available bandwidth is not in stress.

(`Client.Latency*[Cost`$_{latency}$`] > System.Bandwidth*[Cost`$_{bandwidth}$`]`)
`=> <TVListing configuration="FastResponse"/>`

6 Conclusion

We described a new approach for building context-aware Web applications. Using our system, a Web application can adapt to specific contexts through reconfigurable component distribution. Patterns of distribution are extensible: as interesting patterns are identified as useful, developers can define attributes and add new generators that are able to synthesize components with the desired behavior, or just design new specification blocks that realize the patterns. Such extensibility ensures existing Web applications can evolve in the face of widespread change in the Web environment and their users' interaction. The system itself is adaptive: new context features can be integrated by adding the corresponding context variables and monitors, followed by adaptation policies conditioned by these variables.

As future work, we are exploring solutions to automatic policy design and optimization. Currently, good adaptation depends on human design in specification blocks (partial plans) and adaptation policies. As mentioned earlier, a typical Web application is executed numerous times a day and a few bad deployments do not incur much loss. There are great opportunities in mining the performance history and exploring test cases for better policies. We expect future Web applications will adapt themselves automatically by learning their past usage patterns.

References

1. Aspect-Oriented Software Association. http://www.aosd.net/.
2. Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects: Asynchrony-Mobility-Groups-Components*. Springer, 2005.
3. Po-Hao Chang and Gul Agha. Supporting reconfigurable object distribution for customizable web applications. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1286–1292, 2007.
4. Edsger W. Dijkstra. *A Principle of Programming*. Prentice Hall, 1997.
5. Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of ACM*, 44(10), 2001.
6. T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 128, Washington, DC, USA, 1998. IEEE Computer Society.

7. John Robert Gardner and Zarella L. Rendon. *XSLT and XPATH: A Guide to XML Transformations*. Prentice Hall, 2002.
8. Jesse James Garrett. Ajax: A New Approach to Web Applications, February 2005.
9. Google Inc. Google Web Toolkit - Build AJAX Apps in the Java language. http://code.google.com/webtoolkit/.
10. Mik Kersten and Gail C. Murphy. Atlas: a case study in building a Web-based learning environment using aspect-oriented programming. *ACM SIGPLAN Notices*, 34(10):340–352, 1999.
11. Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
12. Wai Yip Lum and Francis C. M. Lau. A context-aware decision engine for content adaptation. *IEEE Pervasive Computing*, 1(3):41–49, 2002.
13. Eric Meyer. *Cascading Style Sheets: The Definitive Guide*. O'Reilly, 2000.
14. NextApp, Inc. Echo2. http://www.nextapp.com/platform/echo2/echo/.
15. M. Philippsen and M. Zenger. JavaParty – Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
16. Arno Puder. A code migration framework for ajax applications. In Frank Eliassen and Alberto Montresor, editors, *DAIS*, volume 4025 of *Lecture Notes in Computer Science*, pages 138–151. Springer, 2006.
17. Arno Puder. XML11 - an abstract windowing protocol. *Sci. Comput. Program.*, 59(1-2):97–108, 2006.
18. Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*. IEEE Press, October 2003.
19. Carlos A. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.