Contributions of the Dichotomic View of Plasticity in the Personalization of Accessible User Interfaces

Montserrat Sendínª, Juan Miguel Lópezª
ª GRIHO HCI Research Lab., University of Lleida, Jaume II St. 69, 25001, Lleida, Spain

Abstract
Recent legal changes have increased the need for developing accessible user interfaces in computer-based systems. In this sense, previously existing user interfaces are intended to be modified and new user interfaces are intended to be designed taking accessibility guidelines into account. Typically, model based approaches have been used when developing accessible user interfaces or redefining existing ones. But the use of static models leads to the development of not dynamically adaptable user interfaces. Dynamic adaptation in accessible user interfaces is important due to the fact that interaction difficulties on disabled people may change through use. In this paper we present some contributions that can be obtained from the application of the Dichotomic View of plasticity in the personalization of accessible user interfaces. With the double perspective defined in this approach, it is intended to go further from a mere adaptation to certain user stereotypes. It is pursued to personalize the user interface by taking into account each user’s limitations, difficulties and particular necessities for systems conceived to be operated in different platforms. This goal is achieved analyzing user logs by an inference motor that dynamically infers modifications in the user interface to adjust it to varying user needs. A case study is presented in order to show how a particular system can be developed using the Dichotomic View of plasticity in order to perform dynamic user interface personalization.

1. Introduction
Demographic importance of accessibility is remarkable. For instance, according to the Eurostat, [1], from a total population of 362 million people in Europe in year 1996, a 14.8% of the population between 6 and 64 years old have physical, psychological or sensorial disabilities. Among themselves, a 4.6% have severe disabilities while a 10.2% have moderate disabilities.

There are also powerful legal reasons in order to develop accessible user interfaces (UI henceforth). For instance, Section 508 [2] requires Federal agencies in the United States to make their electronic and information technology accessible to people with disabilities. This kind of legislative changes have also been made in a lot of other countries around the world, such as European Union countries [3].

It is remarkable that these legal frameworks do not only affect public administration, they also affect corporations that develop software or provide services for their use in civil service. This situation has led to an improvement in the accessibility level of corporation developed software. For instance, Adobe Corporation has been making relevant changes in their software to meet accessibility standards required by the public administration [4]. These changes have affected common use software such as Adobe Reader for PDF files and Adobe Photoshop [5].

In this sense, it can be clearly stated that accessibility support on computer-based systems is a demanding issue. It stimulates developing accessible applications. However, as different types of disabilities exist, the application must be adapted in order to work correctly for as many different
disabilities as possible. In this context, it must be taken into account the fact that disability degrees on people may change through time. As disability degrees can get worse in time, there may be a need for more accessibility features supported in UIs for concrete users. Besides, rehabilitation can make the use of several accessibility features used in the UI unnecessary or disturbing for certain users. Along these lines, the mere fact of developing accessible UIs does not guarantee a dynamic personalization when user needs change in time.

Apart from the merely legal scope, the design of universally accessible UIs has a great positive effect in the extent of socialization of people with disabilities. In this sense, in the scope of Human-Computer Interaction (HCI), the use of inclusive design guidelines is advocated when designing and adapting UIs [6].

UI adaptation to people with disabilities is generally focused on the use of user profiles in order to adjust design decisions in the process of developing the interface. In this line there exist Model Based (MB henceforth) tools that support a progressive UI design in order to gradually integrate different contextual factors, such as the ones concerning users. In any event, some previously typified characteristics are usually treated (such as disability type and degree, preferences, priorities). They allow identifying what stereotype a concrete person belongs to within the people with disabilities collective. Adaptation parameters usually hew to a set of static and system-independent parameters materialized through user profiles. Consequential UI responds to more or less generic disability patterns.

Nevertheless, the UI provided to the user rarely offers any other support for real necessities or difficulties the user meets with while using the UI, such as particular user needs and platform specific restrictions. It is interesting to offer a dynamic support that allows keeping on personalizing the UI. For instance, in order to facilitate user interaction, it can be useful applying small readjustments in disposition, size or other UI characteristics that can be interesting concerning accessibility.

In order to offer enough flexibility to accommodate a specific system to different devices and situations, it is necessary to consider the context under a wide perspective. In a broad perspective, the variability and multiplicity of parameters introduced by all the previous issues (mobility, heterogeneity and adaptation to changing contextual situations) are gathered under the plasticity term. It was coined in 1999 as the ability of systems to mould their own UI to a range of computational devices, conditions and environments in order to tackle the diversity of contexts of use in an economical and ergonomic way [7]. Since then, a great amount of tools have emerged offering a solution to some of these issues.

The Dichotomic View of plasticity approach clearly delimitates this double slope by distinguishing between the runtime support and the design of the UI, in this case adjusted to certain accessibility parameters. In other words, information included in a user profile is complemented by studying and considering the user performance pattern when interacting with the system. As a result of this monitoring, new necessities are inferred, such as identifying what actions can suppose difficulties for the user, in order to propose UI improvements that may overcome them. It is up to make reference to the difference between adaptable and adaptive UIs as complementary properties in the adaptation. This distinction is widely acknowledged [8], [9], [10]. In this paper we present some contributions that can be obtained from the application of the Dichotomic View of plasticity in the support of accessible UIs, by means of: (1) systematization in the generation of this type of interfaces by applying a MB approach; and (2) the incorporation of dynamic aspects that also allow adjusting the interaction process, according to the user behavior pattern during the use.

The paper is structured as follows. Next section discusses some related work. Section three presents the proposed the Dichotomic View of plasticity and the underlying infrastructure. Then, we present the case study used to evaluate the validity of the proposed approach. The architecture proposed for performing dynamic adaptation for the case study is presented next. A brief discussion about possible benefits, some conclusions and further work conclude the paper.

2. Related work

HCI has been concerned for a long time about researching in the scope of accessibility. In this sense, methodologies, guidelines and mechanisms for providing Universal Access [11] have been developed. Therefore, the development of accessible UIs has been within its subject purview. In this direction, not only inclusive design guidelines have been proposed [6], but also tools for working with guidelines have been proposed [12].

The most remarkable example of accessibility related guidelines is the development of the Web Content Accessibility Guidelines (WCAG) [13]. As the use of web based UIs grew, it was clear that it was necessary to create a set of guidelines in order to allow the web to be accessible to users with disabilities. The development of WCAG guidelines was possible due to the use of standards in web environments that allowed providing techniques for effectively transforming non-accessible code into accessible one [14]. Guidelines regarding accessible web content in mobile environments have also been developed [15].

Unfortunately, these guidelines are only applicable to web standard based environments. As a consequence, each no web standard based technology must develop its own accessibility guidelines in order to reach a minimum accessibility compliance level. It leads to the necessity of
providing mechanisms for developing accessible UIs in non web based environments. In this sense, it must be taken into account that solutions provided are specific to the concrete environment where the application is used. For instance, Java Accessibility API [16] allows developing accessible Java applications. As another example, Microsoft offers a series of resources [17] for developing accessible UIs in Windows environments, including specific accessibility APIs such as Microsoft Active Accessibility and UI Automation.

It must be taken into account that not only the development of accessible UIs is pursued. Adaptation of UIs is also a challenging issue. UI automatic adaptation is a long-time research line in the scope of HCI [18], including research in mobile environments [19]. In some cases, UI adaptation techniques have also been applied to research about universal access [20]. It is also remarkable that research in UI adaptation area has included advances in other related areas such as user modelling [21].

As this work is based on the Dichotomic View of plasticity approach, next section describes it deeply along with its background.

3. Proposed Approach: the Dichotomic View of Plasticity

Keeping in mind that our goal is to integrate accessibility issues in a specific infrastructure of plasticity, with the aim to provide systematization, flexibility in the adaptation and a dynamic personalization, in this section the approach and infrastructure of plasticity we consider the most appropriate for our intention is described. The goal is to revise and specialize it for scenarios intervening disabled people.

As already said, plasticity arises as the research area to master the diversity of contexts of use in an economical and ergonomic way [7]. It can be defined as the capacity of systems to adapt themselves –their UIs or functional core– to a great range of contextual factors preserving at the same time a predefined set of usability properties.

In order to provide plasticity in a comprehensive and accurate way, it is not enough tackling the design and construction of new UIs under certain contextual changes, but it is also essential to keep watch over plasticity during runtime, as the user goes through new contexts of use. We are referring to the construction of UIs provided with adaptation mechanisms capable of self-adapting\(^1\), as dynamic variations in the context of use turn up.

In this line, the Dichotomic View of plasticity [10] identifies and delimitates two levels of operation in the plasticity process, which are associated to the stages of reconfiguration–redesign– and execution that the UI should withstand over the whole system’s lifetime due to any type of contextual change. According to that, two types of plasticity have been defined, which make up an extension to the Thevenin and Coutaz concept of plasticity [7]. They are called explicit plasticity\(^2\) –design stage- and implicit plasticity\(^3\) –runtime stage. Simply put, while the implicit plasticity refers to adaptations whose complexity is low enough to be solved in the own platform at runtime, the explicit plasticity involves more complex adaptations that cannot be handled locally. In the last case, the client platform requests a server for an explicit plastic adaptation, which involves redesigning. Otherwise, we talk about implicit plastic adaptations.

To be more precise, explicit plasticity tackles relevant contextual changes pre-established in the design stage that involve the generation or reconfiguration of a new UI, provided that this operation is unsolvable locally in the device (e.g. changes in the computing device or a user replacement). It is responsible for offering appliance-independence\(^4\) and a support for hard adaptations. This stage copes with a lot of decisions about how all the conditions and restrictions involved interact among them. Due to its considerable scope, the explicit plasticity needs to be managed by an external engine –a plasticity server-, where it is brought into operation under an explicit request by the client platform. This is why it is called “explicit”. In this request the conditions of the current context of use are described in order to make them know to the server. This process is depicted in Figure 1.

Implicit plasticity tackles little variations in the context of use, which generally have a dynamic nature (e.g. changes in the brightness level, in the user’s location, etc.). Thereby, it is in charge of providing proactive adaptation at runtime. We are referring to adaptivity\(^5\). It manages either specific modifications in the UI or concrete extra-functional actions. These changes must be solved by an automatic and real time readjustment on the client side, without the express intervention of the user, whenever the own device can assume them. This is why it is called “implicit”. These types of variations are also pre-established in the design stage. In the case of environmental changes, we talk about context-awareness. If these variations are related to user’s needs and preferences, it is used the term personalization.

Clearly both challenges require different modelling, strategies and tools; hence they need to be studied and

---

\(^1\) Applying concrete modifications or little readjustments on the UI on the fly.

\(^2\) Capacity of systematically generating as many UIs as computing environments are needed, starting from an abstract specification. That is, this property has the responsibility for producing a specific UI each time it is required by the client, valid for the context of use at hand.

\(^3\) Capacity of incremental adaptation that the specific UI obtained in the production stage should show at runtime as the user goes through new contexts of use.

\(^4\) Capacity to switch among a set of predefined computing platforms, without loosing execution context, necessary to offer heterogeneity.

\(^5\) According to Dix [22], adaptivity is “the modifiability of the UI by the system”. That is, the adaptation is produced automatically, without the express intervention of the user.
developed in two different frameworks to tackle each goal separately. The division of the plasticity property into two sub-concepts is called a “dichotomy”.

Under this twofold perspective, the infrastructure of plasticity consists of combining two different engines, each one focused on a different goal, framed in a client-server architecture. These engines are respectively called Explicit Plasticity Engine (EPE henceforth) and Implicit Plasticity Engine (IPE henceforth). On the one hand, the EPE consists of a systematic development support capable of generating (sometimes reconfiguring) remotely a suitable UI for each contextual change not solvable on the client-side. As already said, this engine is triggered under a request by the client platform, operating then in a new design stage. We are referring to a plastic UIs development tool capable of adapting the UI according to certain accessibility parameters established in the design stage, which are focused on the disability features of users and that also takes into consideration the current platform characteristics and environmental conditions. On the other hand, the IPE consists of a runtime adaptive engine with the capacity to detect the context and reacting in order to adapt the UI to the contextual variations on the fly. The goal is providing the proactive adaptation pursued on the client side. In this line, our interest is focused on developing a generic framework easily customizable to suitable IPEs for particular systems. This is what is called Implicit Plasticity Framework (IPF henceforth). In the case study in this paper, monitoring impaired people’s behaviour during the use gives also this engine the capacity of identifying which characteristics of the UI generate certain difficulty to the user and consequently applying certain actions in the line of improving the interaction at runtime.

Both engines operate in an alternative, iterative and complementary manner according to the ongoing contextual circumstances and to the evolution of the interaction between the system, the user and the environment. The underlying client-server architecture provides at the same time an appropriate communication infrastructure in order to keep both sides in continuous updating. The goal is to give feedback to the plasticity process without discontinuities. Figure 1 shows the overview of the process described, as well as the delimitation between both sub-concepts of plasticity. The components that characterize the context of use, that is, those that make up the contextual information to be sent to the server are the next ones –from left to right in Figure 1: the environment, the user (current needs and user profile), the platform and the task at hand. This information is taken into account in the server-side in order to produce a customized UI to be returned to the client. In fact, the server processes the requests updating its contextual map (e.g. registering the new user’s needs and preferences) and generating a UI the most adjusted possible to the new situation, which can involve a new user, a new platform, new environmental conditions, any possible combination of them, or simply new user’s needs.

This approach provides and fosters a set of benefits. The most important ones are: (1) an operational balance between both sides, which allows to distribute the charge according to the computational power of the client platform; (2) autonomy to perform adaptivity, thus reducing dependence to the server and possible failures in the communication system; (3) a real time reaction to certain contextual variations in the context of use, contributing so to a proactive (implicit) adaptation; and (4) specific target device adaptations, and then adjusted if necessary to owner APIs.

The Dichotomic View of plasticity is the approach chosen to manage a comprehensive treatment of accessibility issues. In particular, this paper treats on the guidelines established for the construction of IPEs, focusing on personalizing the UI to impaired people. The goal is to design an IPE in the line of promoting a continuous personalization and improvement of the UI during the interaction process, always focused on the evolution of user needs.

With the case of study presented in the next section, it is intended to illustrate all these aspects and identify which are server inferences and which client pre-established actions. The design and code developed for this case study are provided afterwards in section five.

4. Case Study: Feed reader in a company

4.1. Description of the base system

The application in this case study is a feed reader in a company with several different departments. It has been designed to work in both desktop and mobile environments. Its aim is to allow the staff to be informed about all events in each worker’s department. This is particularly useful when some employees are not physically located in company’s building, although it can also be used from desktop computers. Client-server architecture has been used, where the information is maintained in the server and prepared to be served to the client-side through a feed

---

* Modifiability of the UI by means of the system, without the intervention of the user [22].
aggregator. The information, personalized for each user, can be accessed or modified from the client side.

The cost of HTTP request must be taken into account when the application is used from a mobile device as communication with the server.

Despite the fact that this application has been developed to be used in a concrete company, the feed aggregator in the server side can be used for every existing web page with the same client-side feed reader. However, in these cases certain functionalities such as making use of explicit plasticity to adapt some specific types of information would not be applicable.

The basic functioning of the system is to allow the user to subscribe to the feeds likely to become conversant with, in this case the ones belonging departments in the company. The user can get a feed’s address and include it in the application. From that point the server inspects each inserted feed from time to time to check if new information is available. The user is able to see all the items in the feeds he/she is subscribed to. In this case, the feeds were in RSS format.

The application has been developed using Java technology in the client-side. Depending on whether the UI is running on a desktop or mobile environment, J2SE or J2ME technologies were used. On the other hand, as J2ME high level graphical API simplifies the process of creating graphical UIs, Model View Controller pattern has been used in both client and server side applications.

4.2. Scenario

The scenario where the application has been tested is a mobile environment. In this case, the user used both a laptop and a mobile phone to connect to the feed aggregator. In the case of the laptop the UI was developed using J2SE. UI for mobile phone was developed using J2ME, so the possibilities to perform adaptations on it were considerably fewer than in the case of the laptop regarding the limitations that the mobile edition of Java has.

In this case, a user with low vision impairment used the application, so the mechanisms to perform adjustments in the UI were made taking it into account.

Finally, inside the application, the UI where adaptation was performed was the screen for reading news, in this case the items from the feeds provided by the aggregator.

4.3. Stages in the UI personalization process

The UI personalization process was divided into three different levels or stages that are briefly explained next.

1. Get user logs in the application
   The UI performs user activity logging actions since the UI is loaded. User activity is registered by defining event handlers on UI elements.

2. User log analysis
   User activity logs are analyzed in order to determine if a UI adaptation should be performed. These decisions are made when user activity indicates that a potential accessibility problem can exist in the UI.

3. Activate corresponding change in the UI
   Once the need for a change in the UI is determined, the concrete changes are reflected in the UI in order to dynamically perform the required adaptation.

More details the UI personalization stages can be found on sub-section 5.4.

5. Implicit Plasticity Engine: Client-side Accessibility Support Component

This section describes in depth the architecture proposed for the IPE (Implicit Plasticity Engine), that is to say, the client-side runtime adaptive engine aimed to infer improvements over the UI. Once introduced the general structure and the technical approach chosen, the explanations are focused on the particular case of study presented in the previous section.

Finally, some guidelines of abstraction taken into account in the construction of the final Implicit Plasticity Framework (IPF) are reported, in order to obtain an appropriate level of genericity.

5.1. Design Requirements and Structure for the IPE. Special Considerations on Accessibility

Taking into account that our final goal is to develop a generic framework to easily derive suitable IPEs for a particular systems (what it is called the Implicit Plasticity Engine –IPF–), it is necessary to apply the most orthogonal design strategies to satisfy the most challenging design requirements. In particular, the properties that need to be guaranteed are the following ones: (1) transparency in adaptation (monitoring and consequent application of inferred improvements); (2) reusability across different systems, users and disability types; and (3) orthogonality. In particular, orthogonality is essential in order to the adaptive mechanisms be handled independently among them and in relation to the system core functionality. Only that way they can evolve individually, avoiding conflicts and promoting pluggability\(^7\) and reusability. Orthogonality is especially important in those systems where a high number of concerns must be handled and automatically processed, such as in the case of adaptive and/or mobile systems [23]. Even more, it is particularly critical when

\(^7\) Facility to generate different versions of the same application depending on changing contextual needs, as for example different user profiles.
these concerns are sentenced to hopelessly get mixed and spread along the core functionality, due to the way in which they interact with each other. Concerns that despite of using object-orientated programming present these kinds of problematic symptoms are commonly called crosscutting concerns. 

In order to guarantee the three properties indicated above it is necessary to apply a separation of concerns technology, which focus their effort in modularizing and encapsulating problematic extra-functional concerns (sometimes crosscutting concerns). In our case, the crosscutting concerns (the extra-functionality) to modularize are precisely the monitoring and analysis of the user behaviour, the inference of UI improvements and the consequent adaptation of the UI on the fly. They need to be encapsulated and extracted from the core system to obtain the design benefits aforementioned.

Considering all these premises, an IPE is conceived as a software architecture divided into three layers. The extreme ones—the lower and upper—are respectively: (1) the logical layer, which contains the application core functionality; and (2) the context-aware layer, which is in charge of registering the user logs collected and of carrying out accurate analysis by means of an inference engine also located in the platform and represented in this layer. Additionally, this layer also locates the models whose information will be considered along the inference process. All this components are explained in more detail hereinafter.

Finally, the intermediate layer is responsible for applying the changes in the UI inferred, adapting the UI (logical layer) according to the improvement proposals suggested by the inference engine. In other words, this layer acts as a transparent link between the other layers. It is called (3) adaptive layer. To be more precise, in an IPE designed specifically for making easier interaction to disabled people, the adaptive layer has the responsibility of (1) capturing certain parameters related to the user behaviour during the interaction in order to collect the required user logs. That implies monitoring the specific aspects and components of the UI that imply either certain difficulty or special interest to be personalized to particular impaired users. The log information registered will be analyzed by the inference engine. And (2) triggering certain actions focused on improving the user interaction according to the proposals suggested by the inference engine. The UI personalization process applied by the inference engine is based on a gradual scale defined by different levels of adaptation. The idea is to progressively apply the successive levels according not only to the difficulties detected, but also to the consequent changes observed in the user behaviour.

These responsibilities make up the extra-functionality that an IPE designed specifically for personalizing interaction to particular needs and preferences of disabled users must introduce. Equally important is to notice that this kind of extra-functionality introduces what is known by crosscutting concerns.

In particular, following the guidelines defined in [24] for the IPE implementation, the separation of concerns technology chosen for our purpose is Aspect Oriented Programming [25] (AOP). Therefore, the adaptive layer is actually called aspectual layer. The key points in AOP, as well as the reasons why it is adopted for our purpose are exposed in the next sub-section. In this sub-section it is also shown that AOP is an adequate approach to integrate crosscutting concerns in the system operation, which are encapsulated in special program units called aspects. Aspects act in the business code without the latter knows about the existence of the former (the aspectual layer), by means of the particular mechanisms of AOP (see sub-section 5.2 for more detail). Each one of these program units, which assembles the whole treatment of each crosscutting concern, is placed in the aspectual layer. In short, using AOP two of the three aforementioned properties required are guaranteed: (1) maximum orthogonality and modularization, which promotes flexibility; and (2) a totally transparent way to augment the core functionality. The third property (reusability) will be treated in depth in sub-section 5.3.

Regarding to the structure of the aspectual layer destined to accessibility support, taking into account the orthogonality and reusability properties to be preserved, and keeping in mind that our interest is focused on developing a generic framework easily customizable to suitable IPEs for particular systems and, what is more important, for particular user profiles, it is essential to maintain this separation of concerns approach even to decide the necessary components in each layer. In this line, trying to contribute to pluggability in order to facilitate the personalization of an accessible system to different disability profiles, a different program unit (a different aspect) for each concrete type and range of disability is destined. Each one will be focused in a different monitoring strategy and subsequent treatment. In effect, the aspects to be observed in the user behaviour, as well as the actions to be done in favour of the difficulties detected are not the same for visually impaired people that for a user with impaired hearing.

In relation to the context-aware layer, it is in charge of registering the user logs provided by the aspectual layer and of controlling its analysis by means of an inference engine that acts in background. The goal pursued it to elaborate progressive and personalized UI improvement

---

Extra-functional concerns (requirements) that cannot be neatly abstracted in a functional unit using an object oriented programming language. As a consequence, they spread inevitably along (crosscut) most of the modules of a system and tangle the core functionality, with negative effects on reusability, understandability and flexibility.
proposals, which will be showed to the user next time the application is run, in order to accept or reject it.

In this inference process, which is based on rules related to accessibility issues, whenever new data is registered it is checked if some of the rules defined is activated. Depending on the activated rule, UI adaptation proposals may be inferred, always in concordance with a gradual scale defined by different levels of adaptation. Apart from all the user parameters -both static disability features and dynamic user needs-, other types of considerations as the platform and interaction devices restrictions, the task model for the system functionality and some sort of layout description of the UI also take part in the inference process and in the consequent decision-making. These considerations need to be modelled and supported in the platform. As it is depicted hereinafter in the Figure 6, these models are also represented in the context-aware layer, encapsulating that way all the components related to the user log registry and analysis.

5.2. Aspect Oriented Programming: Key Points and Justification

AOP was proposed by Kiczales et al., (Xerox PARC) in 1990, in an attempt to overcome the increasing complexity of software systems, and it was actually named AOP in 1996. It is focused on encapsulating crosscutting concerns, providing modularization to their whole treatment in separated and concentrated program units called aspects. AOP is considered the best technique for implementing crosscutting concerns because it also guarantees a clean separation between functional and extra-functional concerns [25]. In fact, the level of separation of concerns provided by other similar technologies such as reflection is too far from the ideal. Furthermore, AOP manages to augment the system core functionality without affecting neither the software structure nor the code of the underlying system, thus reaching the level of transparency pursued.

AOP includes a set of additional concepts. Let us see them:

**Joinpoint:** An identifiable and well-defined point in the program flow (for instance, a method call, an access to a variable, and so on) to be interfered at runtime in order to either inject new code or change the base code, thus altering the underlying behaviour, such as it is coded in the corresponding advices (pieces of code). This interference in the normal execution flow is applied in a transparent way, that is to say, without the underlying system knowing about the existence of aspects.

In general, the identification of the joinpoints requires a deep knowledge of the base code and as a result, the identification of the joinpoints generates a strong coupling from aspects to the interfered classes. This fact is going to the detriment of making aspects reusable to different systems. Next sub-section presents how this problem related to reusability is tackled.

**Pointcut:** Programming construct to establish the joinpoints that require some type of extra treatment. Pointcuts capture these joinpoints in the program flow and collect their context in a totally transparent way.

**Advice:** The code to execute when joinpoints are reached and captured by the associated pointcuts. We are referring to the code responsible for handling crosscutting concerns, thus augmenting the core functionality. They can be compared to classes’ methods.

Each pointcut is associated with an advice.

**Aspect:** Program entity to encapsulate crosscutting concerns. It consists of a program unit that gathers all of the pointcuts and advices associated, which are related to a certain extra-functional requirement (cross-cutting concern). They can be compared to the structure of a class.

Figure 2 depicts an aspect component (the Z aspect) composed by two pointcuts (X and Y), which pick out three and two joinpoints respectively. These joinpoints interfere different methods of the base classes, as it can be observed in the figure. The effect produced is augmenting the system with the extra code defined in advices (the sety and setP1 pieces of code).

![Fig. 2. Aspect structure and mechanisms to interfere the underlying system.](image)

It is worth mentioning that the modularization described by aspects requires a subsequent process to compose the final operational system. It is the weaving9 stage, to be carried out by a special compiler called weaver, which “injects” code from advices. This is what is depicted in Figure 3.

In relation to the feasibility to use AOP for our particular problem, we would like to bring forward three points: (1) it has been demonstrated that in effect, the monitoring and registry of user logs and the consequent analysis and decisions treatment make up crosscutting concerns. In particular, it has been experienced and studied

---

9The process of combining and recomposing all the different program units (classes and aspects) to form the final system.
the level of tangling in environmental and personalization concerns using object-oriented programming [24]; (2) In this experimental study, all the favourable considerations in relation to the properties of transparency, orthogonality and, above all, reusability in using our technical approach - based on AOP and explained in detail in the next subsection - are made clear in comparison to other approaches, such as pure object-oriented techniques and the use of GoF design patterns [26]. The fact that actually the underlying system is neither altered nor aware of the extra-functionality integration demonstrates that this approach is effectively non invasive (transparency) as well as that AOP definitively separates functional from non-functional concerns in a clean manner (orthogonality). (3) it also has been proven that the use of AOP is a suitable approach even for compact devices with limited resources. Despite there is not a particular Aspectj runtime for J2ME yet (Aspectj and J2ME are the languages used, as it is mentioned in sub-section 5.3), the resulting code is properly supported by this kind of devices. All details about these experimental results can be looked up in [24]. Definitively, it is shown that AOP allows producing low resource-consuming components suitable to augment the underlying system with adaptive mechanisms, which scarcely increase the size of the final code.

![Diagram](image)

**Fig. 3.** The weaving stage.

### 5.3. Technical Approach: Aspects and Metadata Combination

As mentioned above, AOP manages to affect the system runtime behaviour without causing any impact over the existing code, responding in that way to our initial transparency demand. However, it usually requires a deep knowledge of the underlying system (logical layer) to be operative, usually resulting in system-specific designs. To be more precise, the most usual and simple way to establish joinpoints bases their definition in the methods’ names (the so-called method signature) in order to define particular interception points in the program flow. That incurs in important system-dependences. As a result, the aspectual design becomes system-specific and high-coupled, implying a strong limitation in genericity and reusability aspects.

To avoid this strong dependence with the system and therefore to reduce coupling between the aspectual and the logical layer, it is necessary to capture joinpoints in a generic manner. That is why a metadata-based signature is used. It consists of capturing as joinpoints the methods in the core application that carry just a simple metadata annotation expressly supplied. Thus, (1) associating annotations strategically in methods and classes where user monitoring and UI construction is focused and (2) defining joinpoints based on these annotations, the aspectual layer is relieved from this kind of system-dependences. It only needs knowing about the annotations introduced. However, the impact of annotations in classes can be overwhelming - a phenomenon known as annotation clutter [27] - , and it clearly would be to the detriment of transparency. This inconvenience is solved using a special kind of aspect called annotator aspect [27], whose goal consists of encapsulating the annotations to be supplied to the core system. Thus, the annotator aspect represents a declarative section that concentrates annotations, extracting them definitively from the business code and therefore avoiding having to embed them expressly in it.

The benefits of this strategy are clear. On the one hand, metadata annotations allow abstracting aspects in the aspectual layer from the necessary knowledge from the base system, because it is expressed in terms of annotations, instead of in terms of methods’ names. That contributes to de-coupling between layers that therefore convey to reusability, the property not overtaken using AOP only. On the other hand, the annotator aspect encapsulates the necessary knowledge about the core system in a declarative section, avoiding spreading annotations along the business code. It can be understood as an annotation supplier. That allows keeping the level of transparency required for our purpose. In other words, the only “glue” to attach the adaptation mechanisms to the base system is reduced to the annotator aspect. As system-dependences are located in the annotator aspect, changing it would be likely enough to adapt the same aspectual layer to another system also focused on improving accessibility for the same type of disabilities. Precisely, the goal of the generic framework (the IPF) is to reduce the effort to customize any kind of particularities to different domains of application, inference engines, monitoring strategies and UI adaptations without the logical layer requiring recoding at all.

As a summary, this technical approach allows reaching two complementary goals: (1) removing the impact in the core system, and (2) minimizing coupling and dependences from the aspectual layer to the logical layer. Both goals promote reusability to great extent, contributing to obtain system-independent designs. As a result, it can be concluded that this approach is generic (the references to
the underlying system in aspects are made in terms of annotations) and non intrusive.

Currently, this combination of programming techniques is available in the Java world (in particular, in J2SE for traditional desktop systems). As AOP language we have used AspectJ [28], as it provides metadata-based pointcuts.

5.4. Designing the IPE for Our Case of Study

As has already been explained, to design an IPE focused on accessibility issues, the aspectual layer has to strategically trigger certain actions aimed at improving the user interaction by means of the AOP mechanisms. In order to contribute to pluggability in the personalization of an accessible system to different disability profiles, these ones are treated in different aspect units. Thereby, depending on the disability involved in each case, one aspect specially designed for those particular accessibility parameters is destined. As in our case of study takes part a visually impaired user, the aspect that intervenes is called VisionPersonalization, which is specialized in the treatment of visual parameters.

Under the metadata-driven approach presented in the previous section, methods related to either the user monitoring or the application of the UI adaptations proposed by the inference engine, needs to be supplied by a specific annotation. The annotations used are the following ones: @VisionAdaptable and @VisionMonitorizable annotations, respectively. On the one hand, the corresponding aspect needs to define their joinpoints based on its corresponding annotation, as it is depicted in Figure 4-(a). On the other hand, methods in the logical layer (the business code) that need some of these treatments have to be supplied with the appropriate annotations (Figure 4-(b)).

As presented in the previous section, it is used an annotator aspect to encapsulate these declarations, avoiding the consequent unreadability (explosion of annotations) in the business code. Figure 5 shows how the business code –the CoreApplication class- is now relieved from annotations.

In order to give some implementation details about our proposal and the technical approach, it is shown next two pieces of code. Listing 1 shows the annotator aspect for our particular case of study, which provides two annotations. The goal of each line is exposed next during the explanation.
The ScreenVisionMonitorOps pointcut (lines 16-18) intercepts the execution of the method that shows the information of a particular piece of news. As this method has been supplied with an annotation (the @VisionMonitorizable annotation –lines 2-3 in Listing 1), this joinpoint is expressed by a metadata-based signature (line 15). The corresponding advice (lines 23-28) simply captures the user logs consisting in the time the user destinies to read the piece of news. The times captured are then communicated to the inference engine in order to be appropriately registered.

The ComponentVisionMonitorOps pointcut (lines 18-20) monitors the changes of state of the UI component that is object of interest (the component aspect data member –line 7-) by means of intercepting the corresponding event handler. In this case, it is not strictly necessary to use an annotation, due to the code of the joinpoint is not tight to the application. The corresponding advice (lines 29-34) is also in charge of capturing the user logs related to the variation of this component, in order to be communicated to the inference engine.

The VisionAdaptableOps pointcut (lines 22-23) intercepts the particular screen of interest, those that is associated to the task of reading a piece of news. As this screen constructor has been supplied with an annotation (the @VisionAdaptable annotation –lines 4-5 in Listing 1), this joinpoint is expressed by a metadata-based signature (line 22). The corresponding advice (lines 35-42) substitutes the behaviour of the underlying system (the FeedsReader) by the corresponding actions responsible for applying the level of adaptation currently established by the inference engine whenever the corresponding screen is created. These actions are encapsulated in different methods in order to modularize the treatment and also to facilitate its abstraction and reusability.

An enumeration data member in the infEng object is used to distinguish among three different levels of adaptation defined in the UI personalization process. Thus, depending on the captured logs, the system may infer that UI elements of interest –the textField data member in our case of study- are too small for users, so it would propose a first level of adaptation (the ComponentResize value) consisting in making UI elements appear bigger. The resizeComponent method encapsulates this operation. This way it is expected that interaction with UI will improve.

In case user logs still show that user interaction is not good enough, another inference about UI would be raised, in this case consisting in reconfiguring the screen object of adaptation -the PieceOfNewsForm in our case of study-. This decision is indicated by activating the Reconfiguration value and it is carried out by executing the reconfigureForm method. Depending on the capacity of the mobile device, this operation can need an external support to be done. In this case, the UI reconfiguration would be carried out using the MB support located in the plasticity server (the EPE). As has been explained in section three,
this engine is put into operation via request, which need to be adapted in order to let the server know about the current contextual situation. According to that, this request would be prepared with all the information collected in the context-aware layer models: the current user parameters, the device restrictions and the screen layout information. Once prepared, it would be automatically sent to the server in order to request for an explicit plastic adaptation, illustrating then a complete cycle of plasticity. This would be the functionality included in the reconfigureForm method.

A third level of adaptation would be triggered if the engine infers that vision problems are so strong that additional aid is necessary. In this situation, identified by the AssistiveActivation value, some sort of extra functionalities would be raised. These measures would be in form of Text To Speech (TTS) in the J2SE platform. However, the fact that mobile applications developed in J2ME cannot make use of accessibility features provided by mobile operating systems, joined to the lack of a proper Java Access Bridge for mobile operating systems, makes currently not viable the activation of any sort of assistive technology. In these cases, the activateAssitiveTechnology method would consist in modifying certain UI factors or mobile functionalities, such as changing screen brightness in order to improve the UI visualization, or the activation of the vibration to alert the user whenever considered appropriate, relieving thus him from having to fasten on the screen a glance continuously.

Finally, the None value indicates that any improvement measure has been taken yet.

5.5. Further Steps towards the Generic Framework: the Implicit Plasticity Framework

As it has already been mentioned, what we want to offer is a hierarchical library of aspects and classes (the generic framework called IPF), in order to reach pluggability and freedom to incorporate different runtime accessibility support components in each application, or even in each particular use of the same application. We are referring to a generic framework for easily deriving particular IPEs [29]. A wide and assorted library of generic aspects allows providing any static accessible UI with dynamic personalization capabilities in a seamlessly manner. We are referring to a white-box framework, that is to say, a framework that supports extensibility by providing base classes and aspects to be inherited and pre-defined hook methods to be overridden by application developers. This kind of frameworks usually requires application developers to have intimate knowledge of the internal structure of the framework, but they provide a great support for the developer in order to adapt internal framework functionality.

In order to construct this framework, we are applying not only the most abstract, reusable and generic guidelines being strategic in how structuring the hierarchy of aspects to build, according to the different needs that can be found. We are referring, for instance, to the possibility to design different adaptation strategies for a same disability profile. The idea is to apply factorization as much as possible in order to enhance reutilization in the common parts of the code.

Another strategy we are applying to obtain a good designed framework is applying as much AOP-specific patterns and idioms [30] as possible. For example, it is usual that generic aspects tend to have a large pointcut definition. Whenever the pointcut definition is too complex to be understood, it can be made more comprehensible, adaptable and reusable by applying the composite pointcut idiom [31].

6. Discussion

The proposed approach is based on the Dichotomic View of plasticity. This approach is widely used when adapting UIs to different devices and user profiles in different environments [32].

Although the proposed approach can appear to be quite similar to the architecture framework of the conventional adaptive UI, it includes some innovations that make it different in some important aspects. One of the most important drawbacks observed is that, in general, users are only considered through rough and static models, and the dynamism of the context is limited to changes in the physical parameters [33].

Under the Dichotomic View of plasticity approach, user profiles are based on modular components defined according to different impairments or disability degrees. However, perhaps the main innovation is the fact that UI can be dynamically modified using user logs combined with developed context and user models by means of an inference engine. This has been proved by the case study shown in previous sections.

The architecture software presented to the construction of IPEs on the client-side offers transparency in adaptation, no impact over the base application and no coupling with the adaptation mechanisms. These properties provide us the suitable design principles and the appropriate abstraction level to offer a generic framework to dynamic accessibility support. We are referring to a framework flexible, reusable and extensible enough to be adapted to a range of accessibility demands and systems. Thus, for example, if it was necessary afterwards to extend the IPE presented in this paper with other kind of real time constraints to make the system e.g. location-aware, specific aspects for these kind of contextual factors could be added in a new more complete IPE, which would also be embedded in the system. Incidentally, if it was necessary to change the platform to use, recomposing a new IPE combining those aspects specific for the new platform would be enough.
Moreover, developed inference engine can be reutilized in every UI development where J2SE and J2ME technologies are used. Besides, it can be adapted to be used by other technologies by checking the equivalence among Java and other technology UI elements and by including UI elements not present in Java graphical UI libraries.

Although it is not an objective for this paper, regarding the server-side engine, the advantages of a MB approach addressed to the development of accessible Uls can be summed up in the following ones: (1) abstraction: the UI design is not tight to a single disability profile; (2) systematization: the MB approach integrates the UI knowledge with the use of a design method, offering the necessary formalisms to build the UI in a semi-automatic manner (the user takes part in all the stages), alleviating considerably the cost of production; and (3) reutilization: the same system description can be reused to different disability profiles. In fact, MB approaches provide the foundations for code generation and reverse engineering, what contributes to model and UI design reutilization following a user-centred design.

As research in accessibility improves, it is intended to include more accurate information when building the models used. In the inference engine, it is also expected to improve the accuracy of the activation of rules that activate changes in the UI by testing the proposed framework with different users with disabilities in different environments.

As future work it is foreseen to implement the proposed dynamic component for different scenarios. It is expected to derive valid inference patterns for all the described contextual elements based on log analysis and user and context modelling for each new scenario. It is also planned to develop specific transformation methods based on accessibility criteria. These methods, based on a set of rules and models, are expected to be included in an operative MB tool.

7. Conclusions and future work

Assuming that as accessibility support in mobile devices grows, more possibilities for people with disabilities will also be increased, it is recommendable to reuse as far as possible the work already realised to solve problems inherent to dynamic adaptation in the personalization of accessible UIs field.

In this paper, the application of the Dichotomic View of plasticity to the personalization of accessible UIs has been defined, identifying the advantages of this approach and its contributions to the accessibility field. Moreover, the software architecture proposed for the client-side has been applied to a specific case study with a particular user profile and different platforms.

Regarding this architecture, apart from allowing the use of current devices and technologies to develop accessible applications, it is also prepared to take future developments into account. It is expected that the possibility to develop applications by third party developers using accessibility features provided by mobile operating systems will be helpful for the development of more accessible UIs. For example, due to its total lack of any tactile buttons, the touch screen offered by the iPhone is inaccessible to blind people unless some assistive technology, such as automatic UI elements scanning with TTS, is provided to solve this issue.

Acknowledgments

This work has been partially funded by Spanish Government TIN2008-06228 and TIN2008-06596-C02-01 research projects.

References

[2] Section 508
Retrieved on September 2008 from:
http://www.section508.gov/
[3] Article 26 of the EU Charter of Fundamental Rights
Retrieved on September 2008 from:
http://www.adobe.com/accessibility/
http://www.adobe.com/resources/accessibility/tools/vpat/photoshopCS2.html
http://www.w3.org/TR/WCAG10/
http://www.w3.org/TR/WAI-WEBCONTENT-TECHS/


