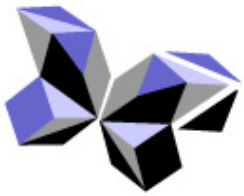


Digital Business Ecosystem

Contract n°507953

Deliverables D20.7: User Interaction Design for DBE Services

Project funded by the European Community under the "Information Society Technology" Programme

Contract Number: 507953

Project Acronym: DBE

Title: User Interaction Design for DBE Services

Deliverable N°: D20.7

Due dates: 30/11/2006

Delivery Date: 13/12/2006

Short Description: This is a report that addresses user interaction design for DBE services. It is written as a part of the deliverable "User Interaction Design for DBE Services" within the Digital Business Ecosystem project. This deliverable is composed of both this report and supplementing code. The code that this document refers to can be accessed at http://dbestudio.sf.net

Authors:

Göran Öberg, Centre for Distance-spanning Technology, Luleå University of Technology

Andrew Edmonds, Intel Ireland Ltd.

Partners contributed: Intel Ireland Ltd.

Made available to: All

Versioning

Version	Date	Author, Organisation
1.0	2006-10-18	Göran Öberg, Centre for Distance-spanning Technology, LUT
1.1	2006-10-25	Göran Öberg, Centre for Distance-spanning Technology. LUT
1.2	2006-11-07	Andy Edmonds, Intel Ireland Ltd.
1.3	2006-11-15	Andy Edmonds, Intel Ireland Ltd.
1.4	2006-12-13	Andy Edmonds, Intel Ireland Ltd.

Quality check:

1st Internal Reviewer: Tim Romberg, FZI

2nd Internal Reviewer: Javier Val, ITA

Table of Contents

Table of Figures

1. Introduction

1.1. Purpose of the report

This is a report based on state of the art in specific areas of user interaction design. It is written as a part of the deliverable "D20.7 User Interaction Design for DBE Services" within the Digital Business Ecosystem project and supplements the work done in the deliverable "User Interface Specification" [66]. This deliverable is composed of both this report and supplementing code. The code that this document refers to can be accessed at http://dbestudio.sf.net. This deliverable has three main goals:

1. This report is initiated in the context of interaction and user interface design and more specifically the design of web-based user interfaces. The state of the art within the two areas of user interface design patterns and computer assisted user interface design will be described.
2. Based on the current state of the art, methodologies are investigated that foster standards and uniformity throughout applications that are built on the DBE software and associated frameworks. Following this, the suggested methodologies are applied to existing DBE applications and evaluated how they benefit from this.
3. From the state of the art, the implementation of tools that will aid developers' productivity in developing user interfaces are investigated.

With these goals in mind, an examination of the state of the art in interaction and user interface design was carried out. Throughout the whole report there has been an emphasis on reusability and this is conveyed to targets of this work, essentially DBE developers, in the form of design patterns.

This work was carried out on the behalf of Intel Ireland Ltd. by Luleå University of Technology, Sweden.

1.2. Background information

From the DBE web site (http://www.digital-ecosystem.org):

> *What is DBE?*
>
> The Digital Business Ecosystem (DBE) is an Internet-based software environment in which business applications can be developed and used. The unique feature of the DBE is that applications within the ecosystem are able to perform new functions that were, up to now, undreamed of by users.

The DBE is an open, free environment where even the smallest specialist software developer can participate competitively in the massive global marketplace for business applications. It will enable end users to easily access and use those applications as services, and to have the benefits of intelligence, interaction and adaptation as the software evolves in response to their own usage and that of others. The initial target of the DBE is those complex commercial transactions and processes that are not easily or economically served by current even state-of-the-art software technologies.

2. User Interaction Review

2.1. Introduction

This review is initiated in the context of user interaction and interfaces and more specifically web-based user interfaces. This section discusses the points of:

- *User interaction* with an overview of the area and its relationship with user interfaces.
- *User interfaces* with an overview of the different kind of user interfaces, from the general interface to the more specific graphical computer user interfaces.
- *Computer and web-based user interface design* discusses the specific area of graphical user interface design where web browsers are used as a common platform.
- *User interface design patterns* introduce the notion of design patterns and describe how they have been applied to user interfaces. The description covers both research done in the area and the commonly used catalogue of languages describing user interface design patterns.
- *Computer assisted user interface design* presents the research in automation of computer user interfaces and also computer assisted development of user interfaces.

2.1.1. Methodology

Background material for this report was found in books concerned with user interaction and interface design and different aspects of usability and from a multitude of different conference proceedings. Conferences that were found to be particularly interesting were:

- International Conference on Intelligent User Interfaces
- Annual SIGCHI Conference: Human Factors in Computing Systems

2.1.2. Assumptions and Limitations

User Interfaces is a broad area which covers all situations where people (*the users*) interact with a machine, device, computer program or any other complex tool (*the system*). A sub area of user interfaces is *Computer User Interfaces*, which in turn contains *Graphical User Interfaces*.

The type of graphical user interfaces that this report will look at specifically, are Web-based user interfaces. What web-based user interfaces are all built with a set of standards described by the work carried out by the World Wide Web Consortium (W3C). Examples of such standards are HTML [39], HTTP [13], CSS [7], XML [28], and DOM [26, 52]. These standards and technologies are described in detail at the W3C website[54].

There are also many other types of user interfaces that are not covered in this report. Examples of these are command-line interfaces, touch interfaces, tactile interfaces, gesture interfaces and physical interfaces.

2.2. Interaction Design

The scope of *Interaction Design* (sometimes abbreviated as IxD, ID or IAD) is the design and creation of interactions between people and any system or structure. These systems or structures may include machines but also organisational structures such as service organisations and companies. An example of the concepts within interaction design is *Interaction Spaces* which are volumes of space that are created by artefacts such as computers and physical objects[35]. The volumes of space define the boundaries within which the devices or artefacts are usable. Interaction spaces can therefore encompass much more than a single computer user interface but also the surrounding that is a part of the environment.

A part of interaction design called Social Interaction Design (SxD) also takes into account the social aspects of interaction design. This is typically seen when computer systems are used for communication and participating in ways that touches on sociology and relational aspects of psychology. As a result interaction design is no longer solely dependent on the classical cognitive sciences that have been traditionally associated with it.

It is often emphasized that the design of interactions needs to be intuitive [4]. Also, interaction design is perceived as intuitive, based not only on the aspect of efficiency, but also on such aspects as how engaging, enjoyable and fulfilling it is. It should also be noted that the content is important for how the interaction is perceived for example, leisure and professional situations put different demands on the mix of characteristics [4].

2.3. User interfaces

The process of interaction design, when applied to any kind of machine or tool, usually results in some kind of user interface.

User interfaces, as seen by the end-users of the DBE framework, are best described as web-based user interfaces. This chapter gives a brief overview and background in some areas of which web-based user interfaces are a part of.

A user interface can be described as the sum of both the physical attributes and the behaviours of a machine, computer system or complex tool that allows a human to interact with it. The interaction may consist of both input and output, for example controlling the system and receiving information from it. Most user interfaces have an interactive component meaning that they can in both a static and dynamic way:

- Receive commands from the user

- Control actions from the user
- Present information to the user

Examples of how a user interface receives commands in physical interfaces could be through knobs, levers and buttons. In the dominating windowing paradigm of computer user interfaces, examples would include the variety of buttons, sliders and text input fields. Information presented to the user can be both static and dynamic. In physical interfaces static information can be exemplified by labels and printed instructions, both graphical and textual. Dynamic information can be exemplified by dials and different types of displays, numeric or analogue. More on the characteristics of computer user interfaces are presented in the following section.

2.4. Computer User Interfaces

Computer user interfaces can also be divided into input components and output components in much the same way as physical interfaces are. Often these input and output components are designed to mimic some characteristics of physical interfaces. Both because the design of physical interfaces often are well known but perhaps because the heuristics applicable often are the same.

The distinction between static and dynamic information presented to the user are often not as clear in computer user interfaces as in physical interfaces. This is because computer user interfaces increasingly use completely dynamic ways of conveying all information to the user. An example of this is ordinary desktop computers, where often the lettering on the keyboard is the only static information shown to the user that can not be changed in such a way that it can show both a mix of static and dynamic information. Most modern computer user interfaces can be divided into three main groupings:

- Personal computers
- Handheld platforms (e.g. cellular phones and handheld computers)
- Embedded systems

In the case of personal computers, the interface is in general divided into a rather standardized set of hardware (i.e. keyboard, pointing device and screen) and the software-based part of the interface.

2.4.1. Graphical Computer User Interfaces

In the case of graphical computer user interfaces, graphical elements as well as text are used to represent information. Currently, the most frequently seen graphical computer user interface is the MS-Windows[1] interface used on many personal computers.

Graphical computer user interfaces are often based on the principle of direct manipulation. Direct manipulation is when users manipulate representations of objects directly by moving and altering them in

[1] Other names and brands may be claimed as the property of others.

a continuous and immediate way. Often real world metaphors are used to help the user build a working mental model in a quicker and more reliable way. Interfaces using direct manipulation often make use of some sort of pointing device. Therefore most graphical computer user interfaces are used with a mouse or similar device.

An important aspect of direct manipulation is the way in which it works as a control mechanism and information mechanism at the same time. The user controls and gives commands by using the same artefacts that gives feedback on the progress, success and the result of those actions.

A typical example of this is moving a file from one directory to another using a graphical computer user interface such as MS-Windows[2]. Consider a desktop with two windows each representing different directories (on the same logical unit for the simplicity of the example), one window empty and the other showing an icon representing a file. The user can then do the following actions, with the accompanying feedback:

1. Placing the pointer above the icon, holding down the mouse button and moving the pointer as an analogue of picking up an object. The feedback from the system is that the icon, or a temporary representation of the icon, is following the pointer in its movements telling the user that the action called “dragging” has been initiated and with which objects the action is operating on.
2. Icon cues given when moving the pointer hint at possible actions. One example of this is when the action is not possible to complete with the pointer at a specific location. The object can then be altered to represent this or the would-be target can be altered to show that it is not a viable target for the current action.
3. When placing the pointer at the intended target and releasing the mouse button, the action is performed and the feedback of this is that the icon should no longer show in the original location but instead is shown in its new location. Variants of this can be different types of animation to strengthen to impression of what the action was and what the effects of it were.

The above example of drag-and-drop is one of the simplest examples of direct manipulation but still consists of several components of control and feedback that is important for the action to be perceived as accessible and easy to understand.

[2] Other names and brands may be claimed as the property of others

2.5. Web-based User Interface Design

This section will introduce the area of web-based user interface design, which is a specific area of graphical computer user interfaces. This is presented as a background to the technologies available to the DBE framework when presenting web-based user interfaces.

When designing web-based user interfaces within the DBE framework using OpenLaszlo [57], either for generating Flash-based interfaces [58] or Dynamic HTML-based (DHTML) interfaces [59] the presented aspects and considerations may be applicable to different degrees. OpenLaszlo hides many of the technical aspects both in presentation and communication but the technologies on which web-based user interfaces are based are still important to know.

2.5.1. Technical Aspects

When designing user interfaces for a web environment there are a number of parameters that need to be considered. The main parameters and aspects are presented in this section.

2.5.1.1. Graphical Design Constraints

Web applications typically have interfaces described in Hyper Text Markup Language[39] (HTML) or its heir eXtensible Hyper Text Markup Language[36] (XHTML). Both HTML and XHTML are mark-up languages that inherit from Standardised Generalised Markup Language (SGML) [22] and define graphical components in more or less structured way. They are a mix of semantic, layout and behavioural descriptions of what should be presented to the user. The layout and behavioural elements are dominant in HTML while XHTML tries to emphasize the semantic aspects to a larger extent.

HTML has evolved over a number of years as a result of the fast paced development of different competing browsers. XHTML is in part a reaction to this and is stricter both on syntax and semantics.

2.5.1.2. State-less

The Hyper Text Transfer Protocol (HTTP)[13] is basically a state-less protocol, with the exception of whatever information is carried in the addresses of resources (most commonly used with the GET method), sent as POST-data (e.g. from a HTML form) and small information containers called cookies that the server can ask the web client to store and send in subsequent requests.

This means HTTP allows for communication and some rudimentary handling of state but does not in itself have any rich mechanisms for managing the state of sessions or applications.

2.5.1.3. Loose Standards

An ever present aspect of web design is the variations in the user environment. Different web browsers behave differently, have or lack support for specific technological details, or have defects that might need to be addressed. This impacts upon consistent appearance and behaviour across different platforms and

browsers[23, 44]. Part of this problem is that the standards from the beginning were loosely defined with respect to presentation, syntax and semantics[53].

2.5.1.4. Multiple Platforms

Web browsers are to a certain degree affected by the environment they execute in. The operating system, its components and applications affect what file formats are supported outside the web browser. The operating and graphical system influences the look and feel of applications and their controls, including the web browser.

In some cases components that are required for a web based user interface (e.g. Flash and Java for applets, QuickTime for media content, etc) can have platform specific quirks and deficiencies that may have an impact on interface performance and user experience. What works in a certain way on one platform may work differently or not at all on another platform or browser. The effect of this in the development process is that if cross-platform portability is important, care must be taken to follow standards and guidelines and that testing on the different platforms performed. To only test functionality on one platform may in some cases be insufficient to ensure portability.

2.5.1.5. Multiple Browser and Variants

Web browsers come in a multitude of different brands, versions and variants. Web browsers can differ in a number of ways:

- Differences in implementation of the Document Object Model (DOM)[26, 52]. The DOM defines the actions possible on the displayed document by JavaScript[11, 31] and other agents in the browser that need to control the document and its appearance. The implementation of the DOM and how to control it often have slight differences between web browsers[24, 42].
- Cascading Style Sheets (CSS)[7] implementations have differences consisting of both errors and deficiencies. Most CSS implementations are incomplete and erroneous in ways which vary between browsers [44].
- There are also examples of differences in JavaScript support between different browsers. This is often a smaller problem, where most incompatibilities in JavaScript stem from differences in the particular implementation of the DOM that the JavaScript operates on, as mentioned above.

2.5.1.6. Proprietary and Embedded Technologies

When designing and constructing web-based user interfaces there is the possibility to employ different types of proprietary technologies that are not described by the standards. They are often supported in web browsers as plug-ins. Their existence in a particular installation of a web browser should not be taken for

granted and there should at least be links to instructions on how to get and install the required software to use the system.

Java[4][18] is one of the most common additions to the regular framework of web standards that are supported by web browsers. The support consists of the ability to run Java applets that are Java programs run in a "sandbox" environment for increased security [17]. The sandbox typically controls access to system resources such as files and networking, allowing the user to allow specific actions, such as saving a file to the local file system.

Flash [58] enables types of applications that are run in a special flash plug-in within the web browser. Flash applets are typically developed in an IDE (Integrated Development Environment) where graphics and program logic are combined into a SWF-file (Small Web-File). Flash technology has its origin in a vector based system called FutureSplash which was acquired by Macromedia [60] in late 1996. Macromedia has developed Flash since then and a large majority of web browsers used has Flash installed, some estimates say as much as 97.7%[1]. In late 2005 Adobe [61] acquired Macromedia including the Flash technology.

2.5.1.7. Web Browser Statistics

To plan for these differences, both in platform, browsers and their varying and shifting capabilities, attempts have been made for a long time to produce reliable demographics on users and their web browser environments.

Web user statistics are difficult because of a number of factors. The identification of web browsers is done using an identification line in the HTTP protocol. This identification is often spoofed, for example in the case of spam bots[3] that often masquerade as regular web browsers[45]. Despite this, it seems that most web browser statistics[41] suggests that the operating systems used for browsing the web is divided between the three largest platforms:

88 %	Windows[4]
3½ %	MacOS[4]
3½ %	Unix (incl. Linux)

[3] A spam bot is a program that collects e-mail addresses from the internet (e.g. ordinary web pages, UseNet, mailing list archives, chat rooms) for the purpose of compiling mailing lists for sending spam (i.e. unsolicited e-mail).

[4] Other names and brands may be claimed as the property of others

2.5.1.8. AJAX - Asynchronous JavaScript and XML

Asynchronous JavaScript and XML (AJAX) [28] is a term describing a technology that has evolved over the last couple of years. It has quickly increased in popularity and consists of web applications that are a tightly integrated mix of server-side programs and programs in the web browser. With JavaScript and advanced manipulation of the DOM[26], an experience that sometimes closely resembles one found in conventional graphical user interfaces is possible. Examples of this are the possibility to drag and drop within web pages and quick updates of information, both graphical and textual, without the need to have pages reload. An important aspect of Ajax-applications is the often asynchronous way of communication with servers. Data such as images and text can be pre-fetched in a way that gives the user a very quick perceived response.

2.5.2. Design Considerations

This section describes methodologies and important aspects to consider when designing Web-based User Interfaces.

2.5.2.1. Navigation and Searching

An important aspect of web based user interfaces is the ability of the user to navigate and find their way around the interface. When a user navigates or searches through a web site, a mental model of the site forms in the user's mind. To facilitate effective navigation and searching, it is crucial to both assist the formation of this model and make it as useful and correct as possible. To assist in the creation of a mental model it is important to have a consistent design and behaviour throughout the web site or application.

To assist in the creation of a useful and correct mental model it is important to think of the users needs when using the web site or application. Differences in user profiles that might be important to target are:

- Variations and differences between first-time users and visitors that have extended experience of the site or application
- Variations and differences in knowledge of the topics on the site or within the application from novice layman to expert of the field.

Designs that are difficult to navigate might be difficult to discover for a developer or designer that are very familiar with the content and functionality the site offers. One way to find these things out is by usability testing. Usability testing is a collection of techniques which, among others, involve having users perform different tasks to find out if their success or failure reveals aspects that need improving[25].

2.5.2.2. Internationalization and Localization

Internationalization and localization are two tightly coupled concepts that are important for user interfaces that will or might be used in international or multi-lingual settings. Traditionally, development of user interfaces and software has often been done with a target on specific markets and customer groups. With the introduction of the World Wide Web, the markets that can be reached and the diversity of potential customers have increased rapidly. Applications and content, both general and web-based, in particular have had a rapidly increasing need of internationalization and localization.

Internationalization is often abbreviated i18n[5] and localization as L10n[6]. The two concepts are linked in that internationalization of a system is the process of preparing it for being localized. Localization is the actual process of adapting a system to local (national) needs in a specific setting. Important aspects of localization are:

- Language translation
- Special graphic and layout support for local writing systems, e.g. logographic, syllabic or alphabetic and glyphs and directions of writing (e.g. Latin, Kanji, Arabic)
- Local customs may affect how content and interaction is best performed.
- Local content may be needed as a compliment or replacement for some of the originally developed content.
- Symbols may either need to be added or replaced depending on local customs or communication traditions.
- Order of sorting may be specific not only to character sets but also to specific languages.
- Cultural values and social context and correspondingly aesthetics.

2.5.2.3. Accessibility

Web accessibility means that people with disabilities can use the web. Use in this context means perceiving, navigating and interacting where all three aspects can be equally important to make use of web systems and content.

Developing web based systems or content in an accessible way can be done in many ways and with at many levels of ambition. Within the World Wide Web Consortium (W3C), the Web Accessibility

[5] The 18 characters between the first and last letters in Internationalization are replaced with 18 and the initial i is preferably written in lower case to avoid confusion when using fonts with similar or identical glyphs for capital i and lowercase L (e.g. Arial).

[6] The principle when abbreviating Localization to L10n is the same as for i18n but here the first letter is preferably written as a capital L to avoid confusion with lowercase i.

Initiative (WAI)[55] has produced a number of guidelines and techniques to help make web development more accessible.

Figure 1 illustrates some technologies and methods for web accessibility and three sets of guidelines developed by WAI:

- Authoring Tool Accessibility Guidelines (ATAG)
- Web Content Accessibility Guidelines (WCAG)
- User Agent Accessibility Guidelines (UAAG)

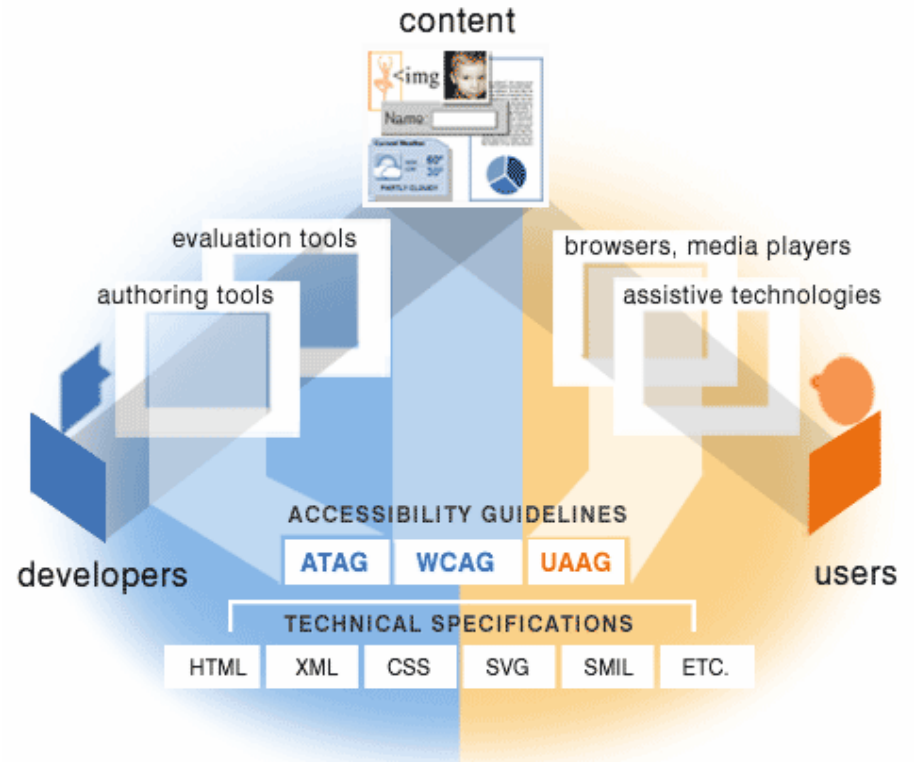


Figure 1: An illustration of methods and technologies for web accessibility (from WAI)

In the same manner as i18n and L10n, Accessibility in the computer domain is often abbreviated as a11y with 11 representing the number of letters omitted. Disabilities can be divided in different groups that have both specific as well as overlapping needs[8].

- Cognitive disabilities can, in functional terms, be described as deficits or problems in areas such as memory, problem-solving, attention, and different aspects of comprehension (e.g. linguistic, math, visual).
- Visual disabilities include blindness and different types of low vision as well as colour blindness.

- Motoric disabilities includes weakness, limited muscular control, limited sensory input, and limitations in movement (e.g. joint problems, missing limbs, impeding pain)
- Aural disabilities include different levels of hearing impairments including deafness.

When designing an interface it should be able to perform well enough, no matter if any sound that is not crucial to tasks, content or navigation, is simply omitted. Comparing the experience for a hearing user with the sound on and the sound turned off might give a hint about the role sounds have been given in the user interface.

A common example of where sound is given an important role is when possible links and action or confirmation of selections and commands are conveyed only with audio cues. In most, if not all cases, visual cues can complement the audio cues to make a more useful interface.

2.6. User Interface Design Patterns

Defining a set of user interface design patterns is useful for web-based user interfaces in general, and for DBE user interfaces in particular. A chosen number of those have been implemented using the OpenLaszlo framework, which is used as an integral part of the DBE framework for developing user interfaces.

This section is intended to serve as a background to better understand the use and applicability of user interface design patterns, both those described as heuristics and those implemented as OpenLaszlo components.

In this section, an introduction to design patterns and user interface design patterns is followed by a discussion of the current state of the art within the area of user interface design patterns.

2.6.1. Introduction to Design Patterns

Before describing User Interface Design Patterns, it is worth mentioning the basics of design patterns as they where first presented in architecture, and later in software engineering.

What is a design pattern? Christopher Alexander, author of the seminal work *A Pattern Language*[2], mainly treating designs patterns in architecture, gives this explanation:

> *“Each pattern describes a problem which occurs over and over again in your environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice”.*

In 1994 Gamma, Helm, Johnson and Vlissides published the book *Design Patterns – Elements of Reusable Object-Oriented Software*[16] which turned into a great success. They presented a concise catalogue of object-oriented design patterns covering a large area of uses along with a discussion of the theoretical background of design patterns.

What are designs patterns not? Design patterns are not off-the-shelf components. Each time a design pattern is implemented it varies a little and even though they may common. Design patterns are not simple rules or heuristics that will walk the designer through the design process in detail.

A complete set of design patterns that covers a whole area is often referred to as "*pattern languages*". The most well known example of this is the pattern language of architecture found in *A Pattern Language*[2] and *The Timeless way of building*[3].

One important aspect of design patterns is the role of a common terminology when describing problems and solutions. Instead of having to describe the context and specifics of a certain problem with a common set of design patterns, or even a complete pattern language, can help communicating situations and propose solutions in a much more efficient way among developers and experts.

2.6.2. Introduction to User Interface Design Patterns

The method of defining a design pattern for a specific problem which reoccurs in an area has proven to be very well applicable to the field of user interfaces.

2.6.2.1. Overview

User interface design patterns are used in a variety of design aspects, spanning from layout, interaction design, accessibility and general usability.

2.6.2.2. State of the Art

The state of the art in the area of user interface design patterns can be described by the current research covering, among other things; principles, requirements, basic conditions and possibilities of the area. It can also be described by the languages of user interface design patterns presented as catalogues covering different domains of user interfaces.

This section is divided into three parts, an introduction and then the description of research and some important patterns catalogues.

2.6.2.3. Introduction

The principles of design patterns are often adapted and modified somewhat depending on the domain they are applied to and what specific goals they are to achieve. To contrast some different approaches to design patterns, these three examples are described:

1. The original work by Alexander[2]. These are the design patterns that all others are modelled from. The design patterns are described with four key attributes: Name, Context, Forces and Solution.
2. Gamma et al in their book, *Design Patterns – Elements of Reusable Object-Oriented Software*[16] (often referred to as the "GoF book" from the abbreviation of "Gang of Four"), adopted Alexander's work to the field of software engineering. Their use of Alexander's model for design patterns was used in a straight-forward way. The examples are, however, aimed at programmers and not at a general audience which is a difference from the original work on design patterns.
3. Tidwell's book *Designing Interfaces*[47] both simplifies and expands on the original way of describing design patterns with an emphasis on graphical user interfaces. The parameters that describe the patterns are: *Graphical illustration*, *Name*, *What*, *Use When*, *Why*, and *Examples*. The "What" is a textual description of what the "Graphical illustration" and the "Examples" are. "What" corresponds to Alexander's "Solution". "Use When" and "Why" are somewhat similar to "Context" and "Forces" in Alexander's patterns. This simpler form is, however, very appropriate for the patterns described. It is to the point, easy to use and is believed that it may very well become influential on future work (Fincher, S., 2006).

2.6.2.4. Current Research

Critique has been made towards the way design patterns have been used in the GoF book. Borchers even went so far as stating in his position paper *Interaction Design Patterns: Twelve Theses*[5] (claim 2), that the GoF book does not contains design patterns, as described by Alexander. Borchers acknowledges the great value of the GoF book in describing a wide range of successful solutions in an inspiring way and that it provides software engineers with a valuable language for describing reoccurring problems and solutions. It does not, however, give non-experts any substantial insight into the problems of and solutions to facilitation of participatory design, which was one of Alexander's main goals with design patterns.

In the short paper *The Gang of Four Are Guilty*[49], Jennifer Tidwell gives her comments on the controversy of whether the GoF book made a bad decision in leaving out one of the original aspects of Alexander's; the participatory involvement of users. She finds that this is a good question to ask and by not asking it, the view of design patterns within the computer industry may have become too narrow. She seems to agree with Borchers that stated, that because HCI is closer to architecture than to software engineering, it has more to gain from Alexander's original principles of design patterns than software engineering has[5] (claim 1).

Borchers also points out that HCI patterns do need to be readable by users in order to fulfil the participatory aspect of Alexander's patterns[5] (claim 8).

An important aspect of the claims Borchers made in his Twelve Theses is that design patterns for HCI need to take temporal structure into consideration[5] (claim 5 and 6). The temporal aspects of Alexander's original patterns are not very prominent. When those patterns have a temporal aspect, it affects basic design decisions, not just the implicit temporal behaviour. One example of this is how changes in traffic flow need to be considered when making geometric decisions.

Published research has, in some parts, focused on different aspects of user interface design patterns and different aspects of using user interface design patterns. In other parts, it has tried to describe languages that span a larger part of the field. One example of a wider approach is *Pattern Languages for Usability: An Investigation of Alternative Approaches*[30] where a study and comparison of four different kinds of design patterns is done:

1. Pattern of tasks
2. Pattern of users
3. Pattern of user-interface elements
4. Patterns of entire systems

They identify patterns of tasks as the kind of design patterns most closely resembling Alexander's patterns. Within patterns of user-interface elements they identify patterns for user-interface element arrangements as a promising target for developing design pattern languages. This is an area that often gets disregarded in the development of user interfaces.

One example of using design patterns to achieve usability both in general and in particular, safety, is A. Hussey's *Patterns for safety and usability in human-computer interfaces*[20] where a design pattern language is described with a explicit focus on usability factors aimed at security. The work built upon and extended earlier work by Leveson[27] and Reason[40] which was presented in the form of guidelines for developing user-design solutions for safety-critical systems. Hussey chooses to describe their design patterns with four elements:

1. Intent is a summary and describes what problems are solved.
2. Motivation describes perspective or facets solved similar to Alexander's Forces.
3. Applicability is a summary of the class of problems.
4. Solution is a set of guidelines assuring safety, based on notion of guidelines as described by Leveson[27] and Reason[40].

From this model and existing guidelines, Hussey compiles a user interface design pattern language that consists of eight design patterns. For conciseness, the design patterns are not illustrated individually but instead the paper presents a single example presenting the patterns combined.

A. Dearden has focused on the participatory aspects of design patterns. In the paper *Using Pattern Language in Participatory Design*[10] Alexander's original approach is used for applying pattern languages for participatory design to the case of user interface design patterns. The original method was characterised by:

1. Remove the separation of architect and builder. The role of the “architect builder” was instead introduced to be responsible for both assisting users in the design phase as well as coordinating the actual building activities.
2. The architect builder helped the users get involved in the design process by first addressing large-scale issues and the dividing users into groups that progressed into looking at smaller scale details and design that affected them.
3. Users were encouraged to make sketches and mark their designs on the actual building sites. This was important for their ability to visualise their proposals and the effects they had on the design.
4. When moving to the actual building activities, Alexander proposed that the process would aim for ”gradual stiffening” where premature commitment to detailed design would be avoided and late adaptations of the design would be possible.

Dearden combines Alexander's methods with previously recognised methods in HCI design and proposed the following:

1. They introduce the role of “designer-facilitator” that helps the user along the design process. This is done by paper prototyping and asking the users questions to make sure they consider different aspects and justify their choices.
2. Design patterns are introduced in different scales and from different viewpoints, for example considering content, general structure and navigational issues.
3. Design prototyping is done mainly using storyboards and paper prototyping. Notes about features and sketches of ideas are encouraged.

4. The design process moves from paper prototypes and sketches to mock-ups and on to designs using web authoring tools and finally gradually finished products. This mimics Alexander's principle of "gradual stiffening". They also refer to other work on "incremental formalisation" within the HCI-area[43].

In their study, they did not focus on developing design pattern languages but tried to use existing languages and existing examples. Their design pattern language was presented as a name, followed by a screenshot showing an example of the design pattern being used and then the textual description of the design pattern, as described by Alexander[2] and also Borchers[6].

One important aspect of the study was the observation that users often did grasp the idea of a design pattern from their language but were perhaps too influenced by the illustration in the design pattern language and tended to treat them as finished design solutions.

Even though it is difficult to discern what could be attributed to use of design patterns and to use paper prototyping working with a knowledgeable facilitator, they found that many of the users were very positive to work with design patterns and found them to be helpful.

2.6.2.5. User Interface Design Pattern Catalogues

Current work on user interface design patterns, in a more applied sense, are perhaps best represented by a number of design patterns compilations. The work takes the form of catalogues describing patterns that are prone to reoccur in designing interfaces for web-based systems and other systems with similar characteristics. The design patterns catalogues most often referenced are:

- *Designing Interfaces* by Jennifer Tidwell (2005)[47, 48].
- *A Pattern Approach to Interaction Design* by Jan Borchers (2001)[6].
- *Interaction Design Patterns* by Martijn van Welie[51].

The book *Designing Interfaces* by Jennifer Tidwell originates from the website *Designing Interfaces*[48] by the same author, which describes an early subset of the patterns published in the book. They both follow the same description model with answers to the questions 'what?', 'use when?' and 'why?' along with graphical examples. Other important examples of design pattern catalogues are:

- *Yahoo! Design Pattern Library* by Yahoo! (2006)[56].

- *Common ground: A pattern language for human-computer interface design* by Jenifer Tidwell (1990)[46].
- *Patterns for Personal Websites* by Mark L. Irons (2003)[21].
- *A pattern language for web usability* by Ian Graham (2003)[19].
- *The Design of Sites: Patterns, Principles and Processes for Crafting a Customer-Centred Web Experience* by Jason Hong and James Landay (2002)[50].

Yahoo! Design Pattern Library is a project aiming at both building a catalogue of web-based user interface design patterns but also to implement support for each pattern in an AJAX library.

Common Ground was the first substantial collection of human-computer interface design patterns. It is relatively large and has been very influential in the field.

Patterns for Personal Websites describes patterns for personal websites and their specific needs but take a rather broad approach describing Content, Structural, Temporal, Navigational, and Technological patterns.

2.7. Computer Assisted User Interface Design

A component of DBE Studio have been developed as part of this deliverable that allows the user to convert the description of services defined in a SDL file (Service Description Language[32, 33]) into a skeleton or prototype user interface. This skeleton is then possible to further edit, extend and adapt to produce a customised user interface for the services offered via the DBE framework. The user interface can either be edited directly in the OpenLaszlo source code or by using the OpenLaszlo Integrated Development Environment (IDE) available for Eclipse. The IDE lets the user work with direct manipulation of the graphical components but is in some ways rather limited compared to direct editing of the source code.

The conversion tool that converts from SDL to a user interface skeleton is a basic form of computer assisted user interface design that automates a part of the user interface design process.

2.7.1. Automated versus Computer Assisted Design

The area of fully automated user interface design is an area of research that still seems to be in its infancy. Attempts at automation of user interface design has had mixed success[38].

It seems the difficulties of fully automating the generation of user interfaces has led the research to focus on automating certain aspects of the design or certain behaviours of the completed design.

Examples of where automation is prominent often require that systems and appliances are modelled in a language and with a methodology that is specifically targeted at generation of user

interfaces[9, 37, 38]. Otherwise, they are rather limited to what types of systems or target interfaces they can handle[34].

Still, some advancement has been made in specifically focused domains, such as automatic generation of forms and automatic generation of database access dialog boxes. Often the level of automation is quite moderate. In most cases the completion of a user interface requires decisions made by a human expert and often in a series of iterations as described in *Adaptation in Automated User-Interface Design*[12]. The principles described in the paper as important guidelines for developing computer assisted user interface design, where parts of the process are automated, are:

1. *Sensitivity*. The algorithms should be sensitive as to require a minimum of input or feedback from the user to influence the process.
2. *Understandability*. The algorithm generating the user interfaces must be understandable to the user. Symbolic methods are therefore preferred over less user-comprehensible methods (e.g. neural networks).
3. *Refinement*. The algorithm needs to be built on current best practices and refine them further rather than start from scratch. This is also coupled to the principle of sensitivity as a system already containing knowledge is easier to steer than one that is empty.
4. *Focus*. The authors do not believe that it is feasible yet, to expect a successful transition from an abstract description of a user interface to an automatically created user interface. Therefore, the principle of focus states that the tools should instead help the designer in those areas where automatic support is possible, such as assisting in the selection of graphical component and producing tentative layouts.

2.7.2. Visual Balance

An example of work done to support the designer in producing a useful and aesthetically appealing layout is *Evaluation of Visual Balance for Automated Layout*[29]. The approach is based on the calculation of the *visual weight* of graphical components. The visual weight is a perceptual notion similar to physical weight and is based on the objects form, colour, texture and placement relative to the visible area.

Also, three established types of visual balance are taken from the visual arts: *Symmetric balance*, *Radial balance*, and *Crystallographic balance*. Symmetric balance is where, given a line through the layout, the objects on either side of the line have equal visual weight. Radial balance is the measure of visual balance around a single point of the user interface and is typically used to create an immediate and obvious focal point of the interface. Crystallographic balance is somewhat the opposite of radial balance

as it strives to have a balance in the layout that does not attract the eye to any given point but rather to the overall presentation of objects.

The result of the paper is an algorithmic approach to calculating the different aspects of visual balance in a layout and also to be able to extract suggestions on how to improve the observed layout.

2.7.3. Automation of User Interface Behaviour

Another approach is the automation of the process of producing variants of a particular interface. An example of this is the automatic or semi-automatic adaptation of user interfaces to different devices or situations[15].

An example of research that has studied automating behaviour of the completed design instead of the design process itself, is presented in *Splitting Rules for Graceful Degradation of User Interfaces*[14]. The paper presents an approach of describing the user interface at different levels, *Tasks and Concepts*, *Abstract User Interface*, *Concrete User Interface*, and *Final User Interface*. They then defines different set of rules to allow automatic splitting and adjustments of user interface components when adapting to smaller or different physical constraints.

3. UI Design Recommendations

3.1. Introduction

This is a selection of user interface design patterns that are aimed at developers of user interfaces for DBE services. They have also been considered when developing the Automatic User Interface Generation Plugin (see 6.1). The design patterns are modelled mainly from the initial design and functionality of the two DBE example applications “Hotel Reservation” and “PC Wholesaler”.

The user interaction design patterns described are compiled from the book “Designing Interfaces” by Jenifer Tidwell [47] and the “Yahoo Design Pattern Library” [56] collection of user interface design patterns.

As the goal is to have the user interface design patterns in the DBE library to be useful to DBE services in general, the design patterns presented here are expected to evolve and change when further studies of converting SDL [32] [33] definitions into user interfaces have been made. The description of design patterns are divided into:

- General User Interaction Design Pattern Heuristics - To be considered whenever designing a user interface.
- Design Patterns planned for DBE User Interface library - Interface design patterns that are planned for implementation in the DBE User Interface Library.

The DBE User Interface library will consist of components implemented as OpenLaszlo [59] classes with a matching framework of Java code to handle the RPC-calls from the user interface.

3.2. General User Interface Design Pattern Heuristics

These are design patterns that do not lend themselves to implementation as separate components or are already represented as existing OpenLaszlo standard library components.

The design patterns described in the referenced literature and already represented, fully or in large parts, within OpenLaszlo standard library components are not covered here as this document assumes a working knowledge of the OpenLaszlo environment.

3.2.1. Diagonal Balance

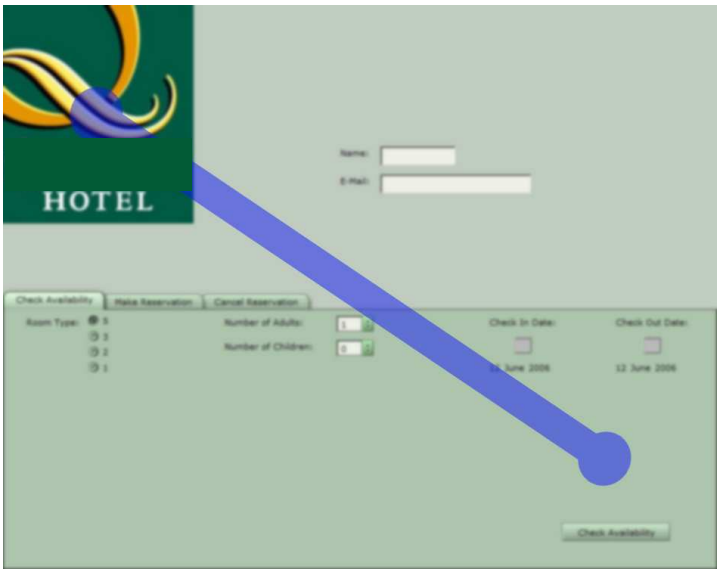


Figure 2 Example of diagonal balance

The design pattern of Diagonal balance describes how arranging visual elements that are vertically opposite also should be horizontally opposite to give the layout a visual balance through a diagonal line. As an example, this means placing tabs at the upper left and the action button at the lower right.

3.2.2. Right/Left Alignment

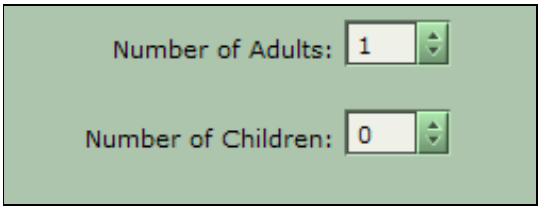


Figure 3 Example of Right/left alignment

The labels should be placed as close as possible to the data entry area they describe. This means that labels in front of selections should be aligned to the right. This design pattern is described at page 116 in Designing Interfaces [47]. The same principle means that the textual representation of the current state of a date selector component should be placed immediately below its label (see ***Figure 3***).

3.2.3. N-radio Buttons vs N-item Dropdown List

When choosing between the components for N radio buttons and N-item dropdown list (***Figure 4*** and ***Figure 5***) the following pros and cons need to be considered.

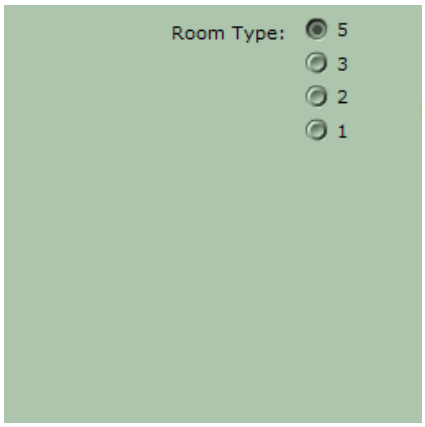


Figure 4 Example of N radio buttons

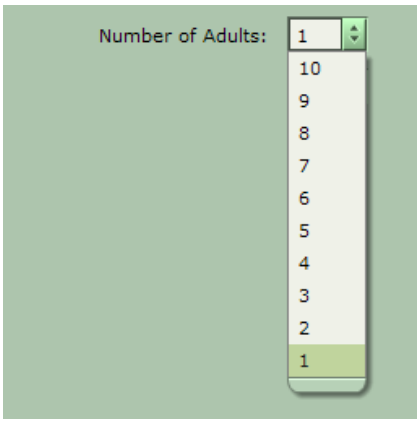


Figure 5 Example of N-item dropdown list

N-radio Buttons Characteristics:

- Pros: All choices are visible
- Cons: High space consumption

N-item Dropdown List Characteristics:

- Pros: Low space consumption
- Cons: Only one choice is visible at a time, except when menu is open. Requires more attention to scroll through and select an item, especially if all items cannot be shown at the same time (i.e. the dropdown list is scrolled).

3.2.4. Prominent "Done" Button

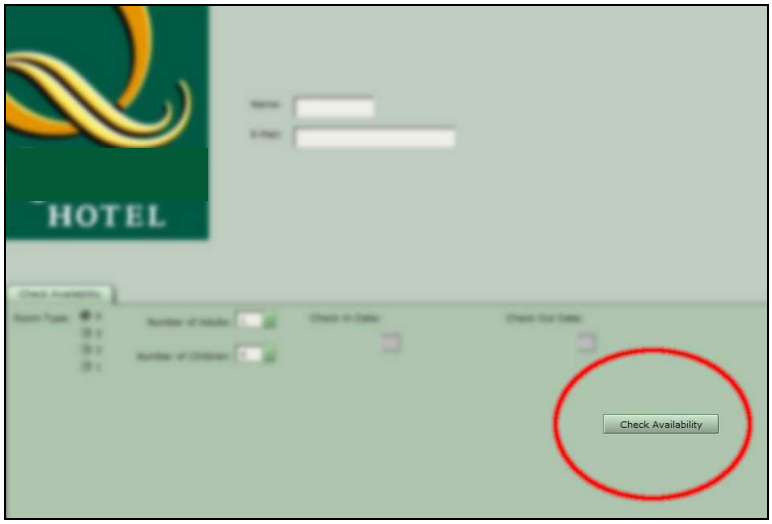


Figure 6 Example of Prominent "done" button

A Prominent “done” button is placed at the end of the visual flow, clearly labeled, sufficiently large and distinctively separated from other components in the layout. This design pattern is described at page 144 in Designing Interfaces [47].

In both user interface examples this means the action button should be placed in the lower right of the layout.

3.2.5. Clear Entry Point

Figure 7 Example of Clear entry point, google.se

Figure 8 Example of Clear entry point (from Designing Interfaces)

The clear entry point design pattern states that a limited number of entry points into the interface are preferred and that they should be task oriented and descriptive. The design pattern is described at page 64 in Designing Interfaces [47].

In the PC wholesaler example user interface this means that the entry point previously only consisting of the title Product Categories should be expanded into categories when the start page is displayed. This makes it a more descriptive entry point.

In the Hotel reservation example user interface the choice of entry points is not as obvious. There are the selection of tabs, the data entry fields in each tab view and the fields for name and email address. All those could be perceived as entry points. An improvement to consider might be to integrate the fields for name and email address to present fewer potential entry points to the user.

3.3. DBE-specific User Interface Design Patterns

A selection of user interface design patterns have been implemented in the DBE User Interface library (see 4.1) and are presented below.

Although the selection of user interface design patterns and user interface components have been chosen based on the two user interface examples PC Wholesaler and Hotel reservation, the patterns are targeted at general services and e-commerce scenarios.

3.3.1. Cascading Lists

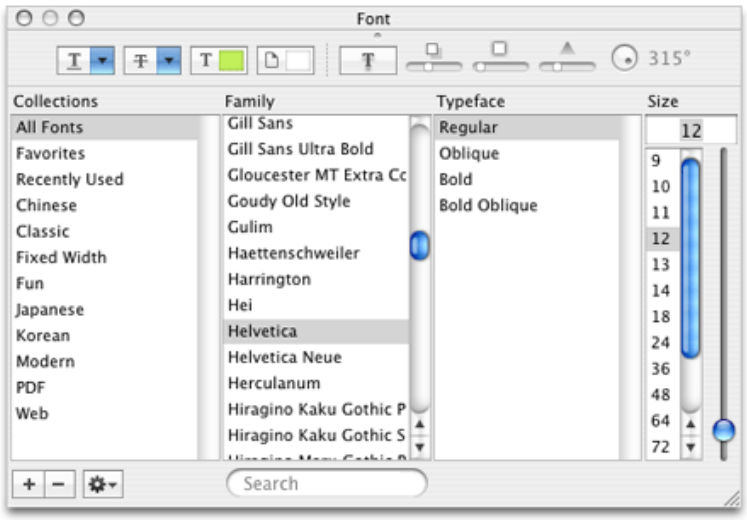


Figure 9 Example of Cascading Lists (from Designing Interfaces)

The hierarchical organization of items is best handled by the Cascading lists user interface design pattern described at pages 195 and 212 in Designing Interfaces [47]. An example of a cascading list is shown in ***Figure 9***. As the hierarchy in the PC wholesaler user interface example is two levels deep, a single-selection tree (the variant described at page 212 in Designing Interfaces [47]) is likely to be the best as it conserves horizontal space. If the overview at the top level is paramount a variant more like the multiple panes at page 195 in Designing Interfaces[47] would be preferred.

3.3.2. Same-page Error Message

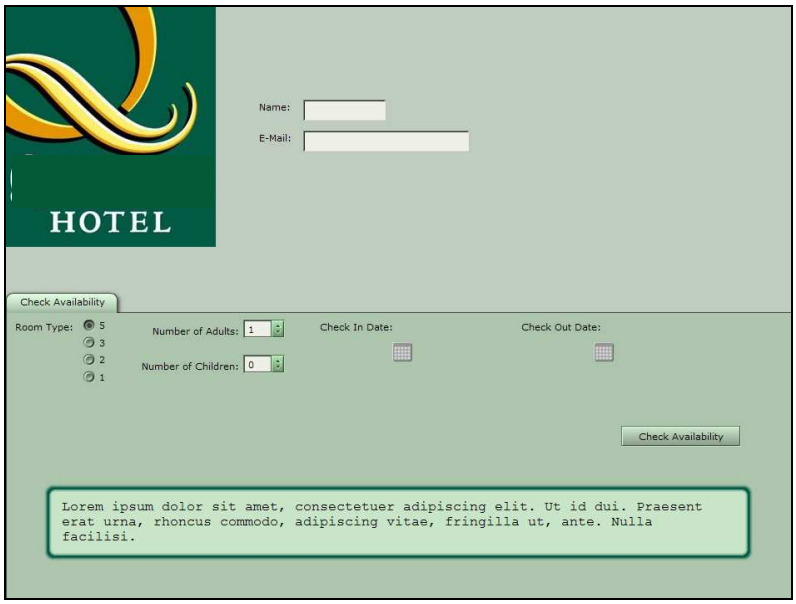


Figure 10 Example of Same-page error message

The principle of Same-page error messages is that of showing feedback and error messages without removing or concealing the information the user was shown when performing the action that led to the message. It also prescribes highlighting any areas referred to in the error message and preferably placing the message at the top of the screen. The placement needs to be in harmony with the overall design while at the same time attracting sufficient attention to alert the user of the message. A message area to display error messages could also display messages about and confirmations of other actions, such as the cancellation of an order. This pattern is described at page 239 in Designing Interfaces [47].

3.3.3. Sequence Map

Whenever a recommended (or mandatory) sequence of actions needs to be presented the Sequence map design pattern is useful. It is presented at page 76 in Designing Interfaces and consists of a visualization of a sequence of actions and a visual cue that shows where in the sequence the user currently resides. The sequence of actions that could be used in the PC wholesaler user interface example is:

1. The selection of an item.
2. Reviewing the selection of items
3. The confirmation of purchase.

The 1st step should be possible to repeat until the user is satisfied with the selection of items. A presentation of this could be done using three tabs, named “Select items”, “View cart” and “Place order”.

At the second tab a button named “Select more items” or something similar should be present. The reason for this button is to make it clear to the user that this repetitive course of action is possible.

In the Hotel reservation user interface example a similar approach could be useful, leading the user from checking for available rooms to reservation of rooms and finally submitting the selected reservation.

3.3.4. Login Form

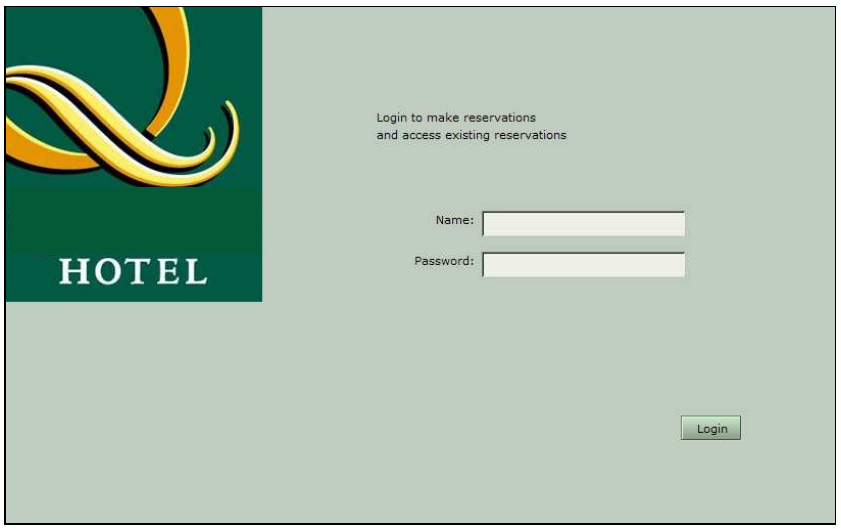


Figure 11 Example of a Login screen

The login screen is a reoccurring design that needs to be consistent with the rest of the user interface design. Preferably there are graphic elements that help the user recognize to what the login will lead to.

3.3.5. Registration Form

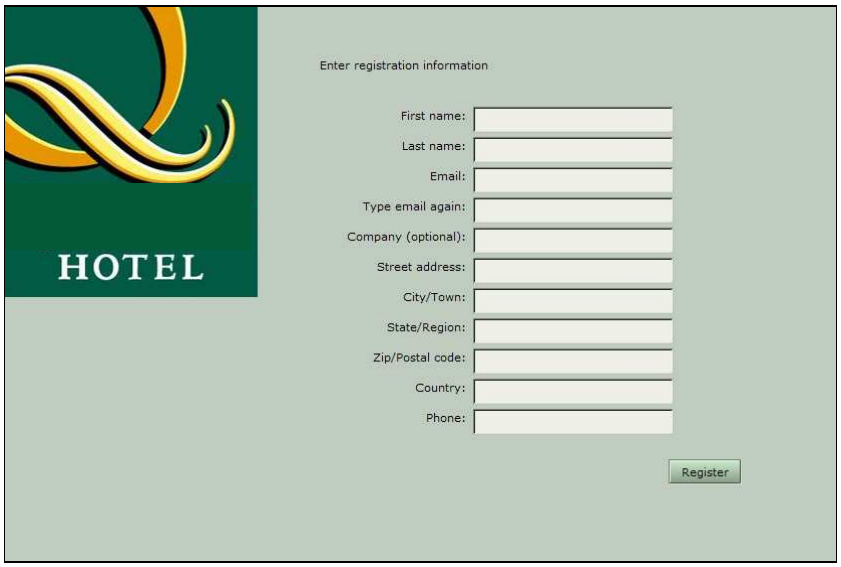


Figure 12 Example of a Registration screen

As well as the login screen the registration screen benefits from a consistent design that lets the user recognize to which system and service the registration belongs.

3.3.6. Info Summary Page

Figure 13 Example of an Information screen

The information screen may be used in many different situations such as presenting detailed views on existing reservations, orders and deliveries. It can also be used to preview information in detail before submitting an order.

4. UI Design Pattern Library

4.1. Introduction

This section describes a number of user interface design patterns implemented as components in OpenLaszlo.

Based on examples from the DBE Studio (version 0.2.0) the following user interface design patterns were selected for implementation:

- *dbe_panel*, a component that provides a generic layout panel with space reserved at the bottom for the display of status and/or error messages
- *dbe_infosummary*, a component providing a simple layout for pairs of labels and text-pairs.
- *dbe_breadcrumbs*, a component providing a representation of a location in a hierarchical structure by showing the path taken from the top of the structure. It is called “breadcrumbs” because of the way it mimics leaving a trace when navigating deeper into a structure.
- *dbe_register*, a component providing a simple layout for pairs of labels and input fields for registration with a service.
- *dbe_login*, a component providing a simple login-screen.

4.2. Components

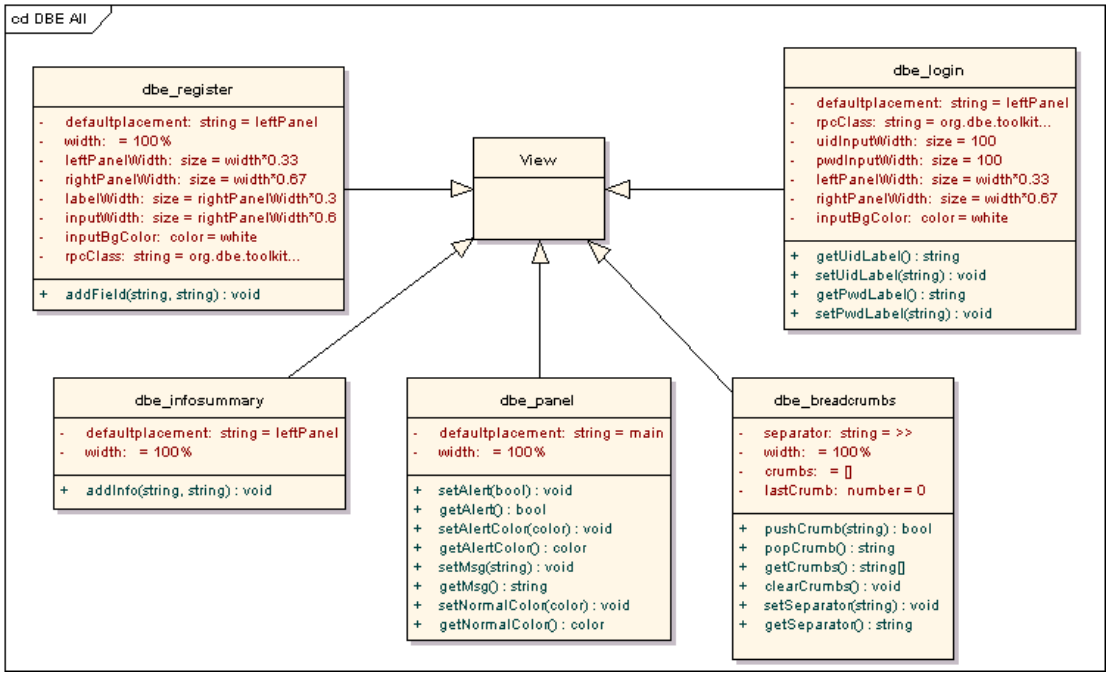


Figure 14 UML diagram of components in DBE OpenLaszlo UI Library

Figure 14 shows the five components of the DBE OpenLaszlo UI Library, all extending the OpenLaszlo Standard Library class *View*. Each component is presented with further comments below.

4.2.1. dbe_panel

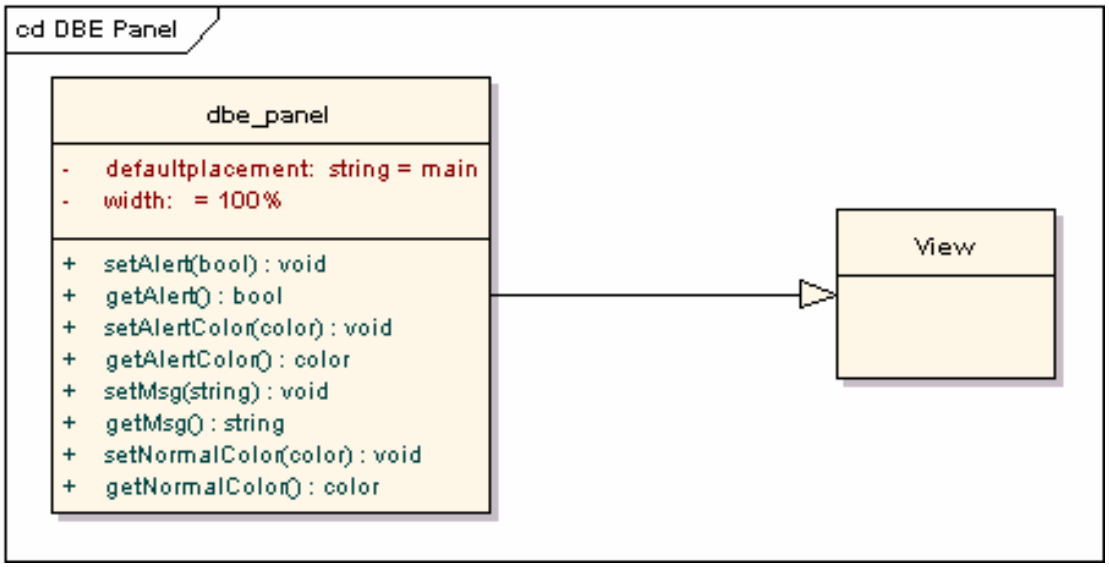


Figure 15 UML diagram of OpenLaszlo component dbe_panel

dbe_panel provides a generic layout panel for a user interface where an area at the bottom is reserved for status and error messages. The upper part of the panel is to be used as a generic view, containing whatever user interface needed. The internal state that controls the content and appearance of the message area are:

- *Msg* is the string of the displayed message. Set to empty string to clear the messsage area.
- *Alert* is a boolean value controlling whether the message area should be highlighted with a frame in contrasting colour.
- *AlertColor* is the colour to use for highlighting the frame around the message area when *Alert* is set.
- *NormalColor* is the normal colour to use for the frame around the message area when *Alert* is not set.

Each of these are controlled using conventional getters and setters, resulting in the following methods: *setAlert, getAlert, setAlertColor, getAlertColor, setMsg, getMsg, setNormalColor, getNormalColor*.

4.2.2. dbe_infosummary

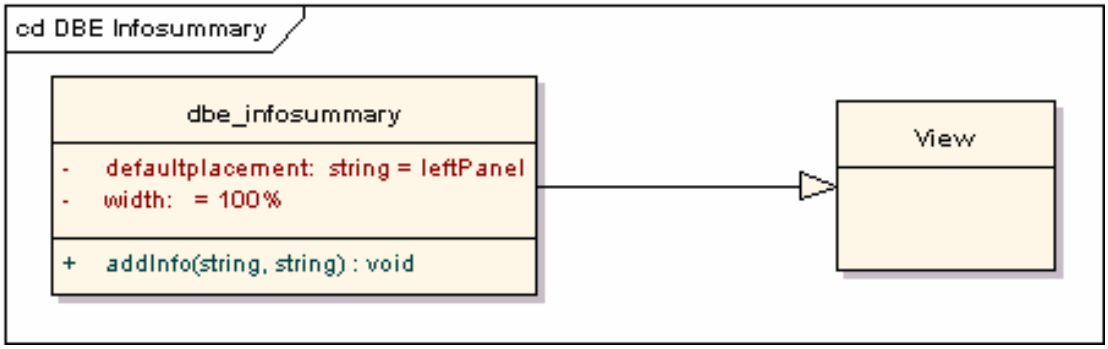


Figure 16 UML diagram of OpenLaszlo component dbe_infosummary

dbe_infosummary provides two panels, one to the left which contains whatever content the *dbe_infosummary* component is given and one to the right where information is placed using the method *addInfo()*.

The information is described with two strings, one label and one information field which are placed together on a row. The collection of lines is aligned vertically in the middle to easily accommodate for different amounts of information.

4.2.3. dbe_breadcrumbs

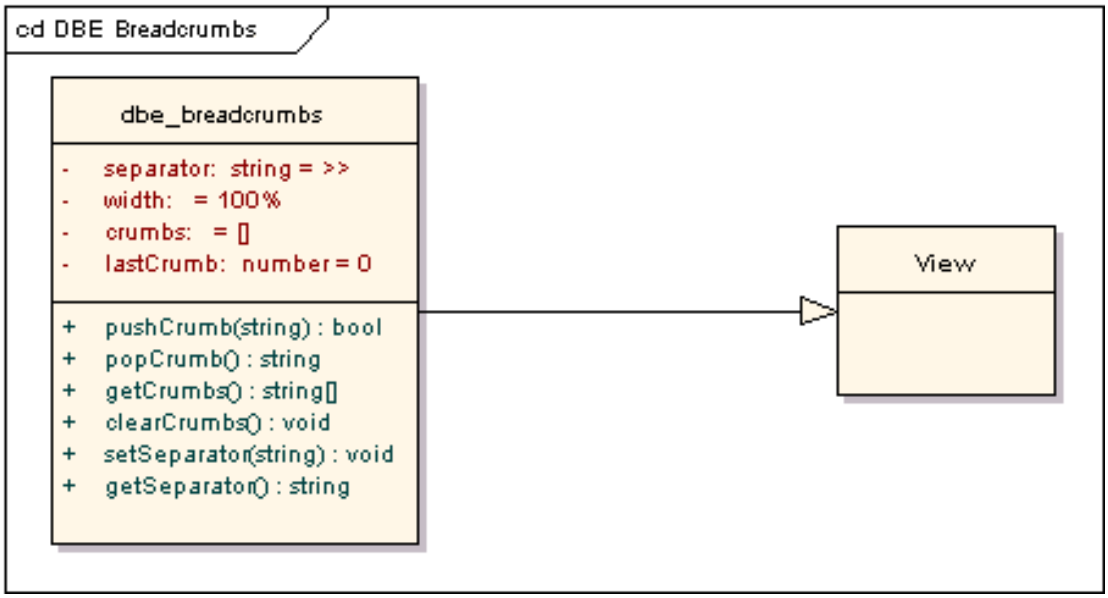


Figure 17 UML diagram of OpenLaszlo component dbe_breadcrumbs

dbe_breadcrumbs provides a convenient way to manage a type of "bread crumbs" that can help the user to know where in a potentially deep hierarchy they currently are. The component offers a number of methods to manipulate the sequence map:

- *pushCrumb(string)* adds the string to the end of the sequence.
- *popCrumb()* removes the last "crumb" from the sequence.
- *getCrumbs()* returns an array of strings with all the crumbs currently displayed.
- *clearCrumbs()* clears all crumbs from the display.
- *setSeparator(string)* sets the separator used to separate the crumbs in the sequence. ">>" is used as default.
- *getSeparator()* returns the currently used separator.

4.2.4. dbe_register

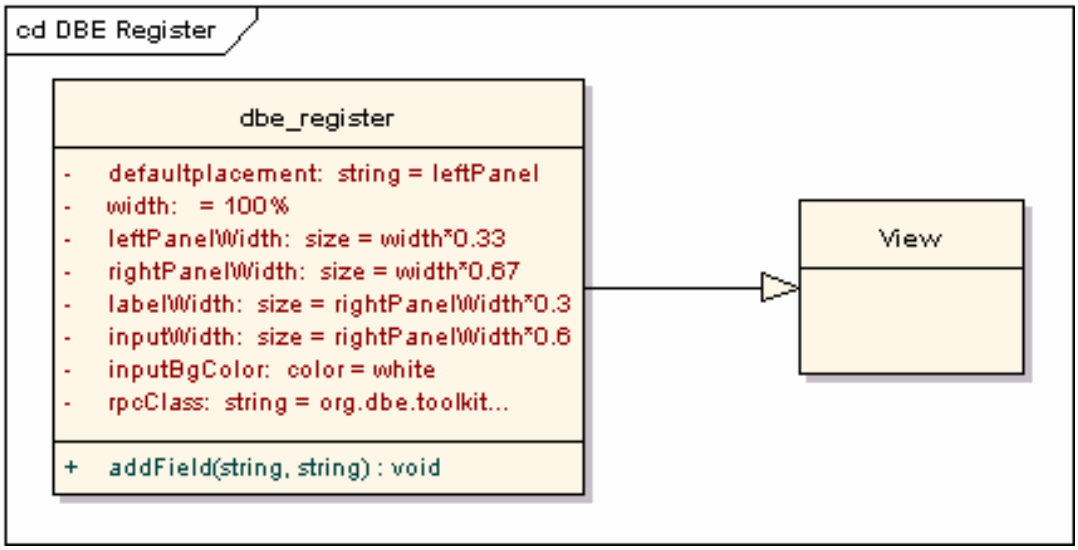


Figure 18 UML diagram of OpenLaszlo component dbe_register

This component allows a user to dynamically build a registration page with components that are supplied by the user. For each particular piece of information that the user is expected to supply, a call to the method *addField(label, name)* will add a line with a label and an input field. The string label will be used as label and the string name will be used to tag the input field. This name may be set to match field names in the registration services thus enabling an automatic mapping from input field to the procedure call for registration.

Since the login and authentication mechanisms within the DBE framework are not yet implemented this component is lacking the actual code to do the authentication. There is a comment in the code where that code should be put once specifications are set.

4.2.5. dbe_login

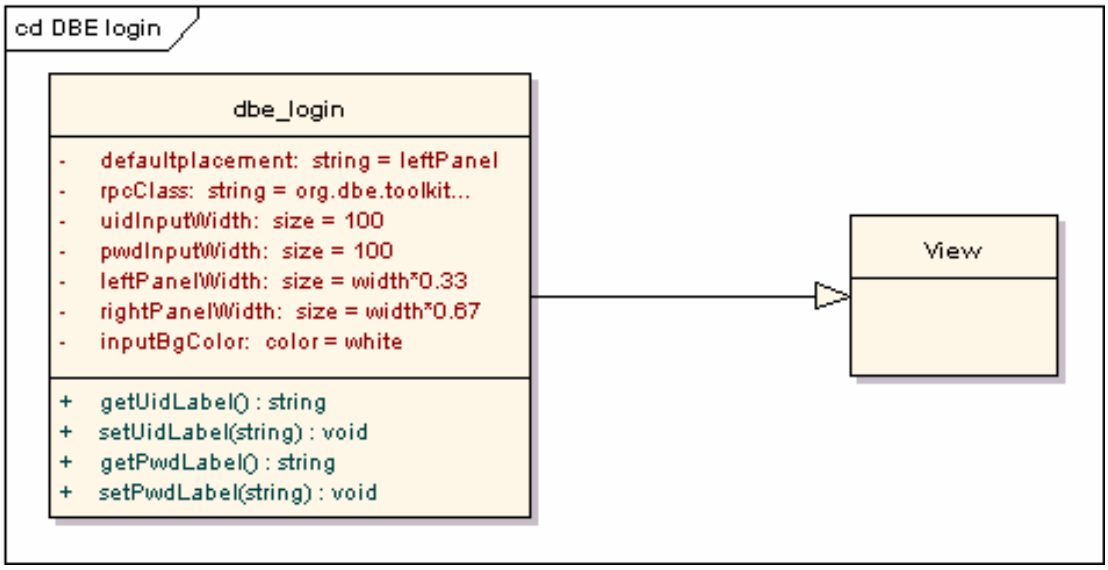


Figure 19 UML diagram of OpenLaszlo component dbe_login

dbe_login presents a basic login dialog with two entry fields, each with a label. The first field is for user name or user identification and the second is for entering a password for authentication. The labels defaults to “User name:” and “Password:” and can be altered with the getters/setters *getUidLabel*, *setUidLabel*, *getPwdLabel*, and *setPwdLabel*.

The class has three instance variables, two that controls the width of the input fields and one that sets the RPC class name to address.

Since the login and authentication mechanisms within the DBE framework are not yet finished this component is lacking the actual code to do the authentication. There is a comment in the code where that code should be put once specifications are set.

4.2.6. Integration with OpenLaszlo

The components have been developed using OpenLaszlo server version 3.3.3. In this version of the server, the components are intended to be installed in a new directory “`dbe`” in the directory “`Server\lps-3.3.3\lps\components`”.

This will let OpenLaszlo programs include components using the prefix “`dbe/`” (e.g. “`<include href="dbe/dbe_infosummary.lzx" />`”)

To be able to include the components without the directory as a prefix there is the possibility to edit the library include scripts in the components directory and add one in the dbe directory but as this means altering the OpenLaszlo server environment the simpler procedure proposed is recommended. It is

also easy to maintain different versions of the components in different directories for development purposes, without the need to restore any settings when finished.

4.3. Anomalies in OpenLaszlo

Large parts of the work programming these components have been related to working around or localizing and avoiding strange behavior in the OpenLaszlo environment.

Most problems have been with the layout. The simple grid packer *simplelayout* often shows much unexpected behaviour when small adjustments are made to surrounding or contained components. In other situations components were rendered with zero sizes or overlapping each other even though the layout should have placed them at separate locations.

The problem with a less than completely reliable and robust simple layout mechanism is that these components need to be simple and generic rather than fixed and rigid. When component size and placement need to be calculated rather than laid out with simple rules there is always the risk that unforeseen situations or uses of the components will break its behaviour or layout characteristics.

One specific example is when two simple view objects are laid out using *simplelayout* in a vertical direction. The upper view has no explicit height set but will instead adapt to the geometry of the contents. *simplelayout* is then unable to keep the views separate but instead lets them overlap. One of the components had this problem and the only solution found was to explicitly give the first view a fixed height.

Another specific example is how alignment of *text*-components works in completely different ways depending on the text being provided as content (via the xml-tag text) or as a parameter. The documentation of OpenLaszlo suggests that it is interchangeable but in reality the resulting layout varied significantly.

4.4. Future Work

During the development of the components some thought on future work have been gathered which are presented here.

4.4.1. Framework Orientation

Instead of creating concrete components that are meant to be used as finished components, it might be a more rewarding exercise to produce a framework where ideas from user interface design patterns are expressed as skeleton architectures and provide suggestions on how to build a useful interface from those skeleton architectures. Completed components risk distancing the developer from the decisions and adaptation that often is seen as an important aspect of working with design patterns.

4.4.2. Feedback from Users

Any use of these components will most likely produce feedback on their usefulness and on what they are lacking and what is not needed. Their use might of course also lead to suggestions for completely different components or classes of components.

4.4.3. Modeling from More Examples

These components were selected based on a set of two example GUIs with a broader DBE perspective in mind. When DBE come into broader use there will be a large number of examples that might point to new needs and ways to use the DBE framework. Those new developments could of course be used as input for any further work on a library of components.

5. Case Study: Applying UI Design Recommendations and Patterns

5.1. Introduction

This case study describes the update of the two example user interfaces found in the DBE Studio [62], Hotel Reservation and PC Wholesaler. In this section we apply the previously discussed user interface design patterns. This includes both those expressed as general design pattern heuristics and also those design patterns available as components implemented in the DBE OpenLaszlo UI Library.

5.2. Hotel Reservation Example

The Hotel Reservation example consists of three tasks:

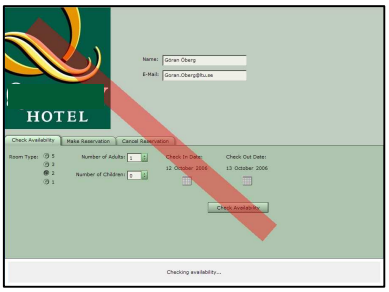
1. Check the availability of hotel rooms
2. Make reservation
3. Cancel an existing reservation

The three tasks are presented on separate panes which are selected by tabs.

5.2.1. Applied User Interface Design Patterns

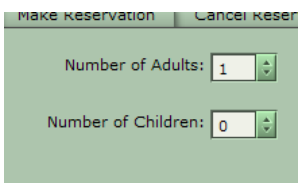
The general changes to the user interface when applying user interface design patterns were:

- *Diagonal balance*. By aligning the prominent objects at the top to the left and aligning objects at the bottom to the right a diagonal balance is achieved. This diagonal balance is often considered aesthetically pleasant. It also supports the notion of a logical flow and direction of order of sub-tasks when solving the task of entering and selecting data in the form. This ends with a button to activate the form and finish the task.



- *Left-aligned tabs.* As a consequence of the user interface design pattern *Diagonal balance*, the tabs are preferably placed to the left to improve the diagonal balance. Having them to the far right would break the balance.

- *Right aligned labels*. By right-aligning labels and putting them close to the field where the text is entered or choices are made, the mental process is set of connecting the label to the place where data is entered or selected. This helps the user to make quick decisions and decreases the risk of errors.



The user interface design patterns that were already present in the Hotel Reservation example user interface were:

- *Card stack*. To support a multitude of tasks within one user interface the content can be placed on cards organized as a card stack. This pattern needs for the content to be logically divisible into the different cards, either as tasks or thematic. In the Hotel Reservation user interface example the division in three distinctive tasks lends itself very well to separate cards in a card stack.
- *N radio buttons*. When selecting from multi-values where the values need to be visible simultaneously, the use of radio buttons is a good choice. An example of when constant visibility might be warranted is when a numerical value is to be chosen from a non-continuous sequence of choices. The drawback is that it consumes more screen space than a dropdown list.
- *N-item dropdown list*. This is an efficient way to save space when presenting the use with a multiple choice input. It is especially suitable when the choices are continuous so that the user is better able to build and maintain a mental model of the available choices without having them visible all the time.
- *Calendar control*. The calendar control used in the Hotel Reservation user interface example is a generic solution for selecting a date using a calendar showing one month at a time. In OpenLaszlo this is presented as an icon that expands to a calendar when selected. In OpenLaszlo (v3.3.3) it is regretfully not possible to have it always expanded without triggering a bug. The bug occurs where the calendar control is permanently lost from the user interface if the upper right corner button with an ‘x’ in it is used to minimise/close the calendar panel. In many cases the always expanded calendar would be preferred as the area saved by the icon is rather small when compared to the clarity lost and the drawbacks of a changing interface. The combination of icon and expanded calendar also introduces one extra action that is required to switch between the two.

- *Single line text field*. The single line text field is obviously suitable for short strings of text and should preferably be wide enough so that the entered text is visible in its entirety all the time. Having the field show only part of what is entered forces the user to allocate more attention to the input and verification of the information and increases the risk for errors.
- *Prominent “done” button*. The button on each card that activates the form and submits information is somewhat oversized and located so that it is easy to find. The upgraded version

places it to the lower right of the form rather in the centre to follow the principle of diagonal balance.

5.2.2. Other Changes

Other changes that were made, not matching a specific user interface design pattern:

- *Improved size of fields.* The input fields for name and e-mail were too short for the expected lengths of entered text to fit.

5.2.3. UI Tasks

The different tasks that the user interface makes possible are presented with screenshots showing the user interface before and after upgrading with the user interface design patterns mentioned above.

5.2.3.1. Check Availability

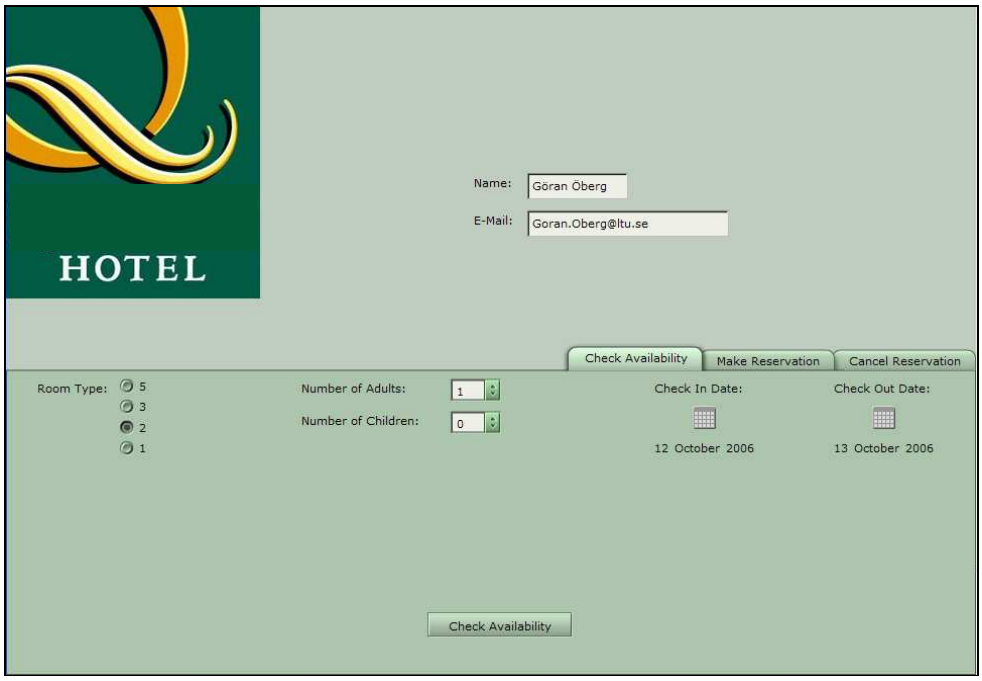


Figure 20 Original example, Hotel Reservation, checking availability

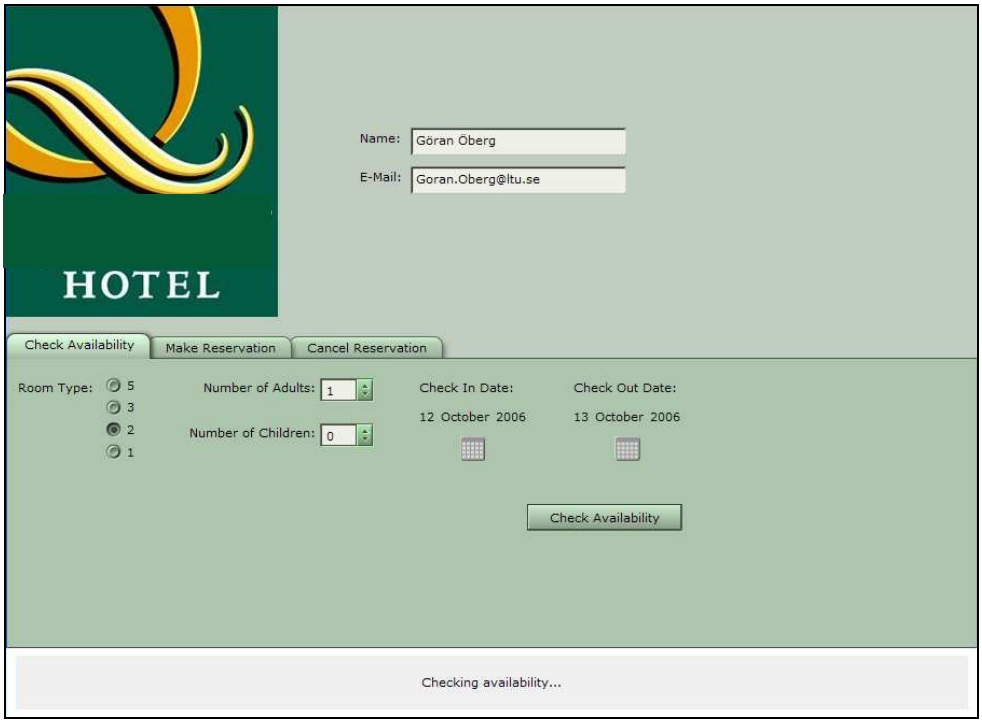


Figure 21 Updated example, Hotel Reservation, checking availability

This is a generic example of where the details of a reservation is entered and the system can be queried on the availability of rooms matching the description.

As is shown in ***Figure 21***, the message area shows the message "Checking availability…" as a result of the user activating the form with the "Check Availability" button. As the user interface is programmed using asynchronous queries to the underlying parts of the system, it would be suitable to give messages that inform of any errors or timeouts that may occur and give suggestions on the best actions (e.g. retry, try later or contact support).

5.2.3.2. Make Reservation

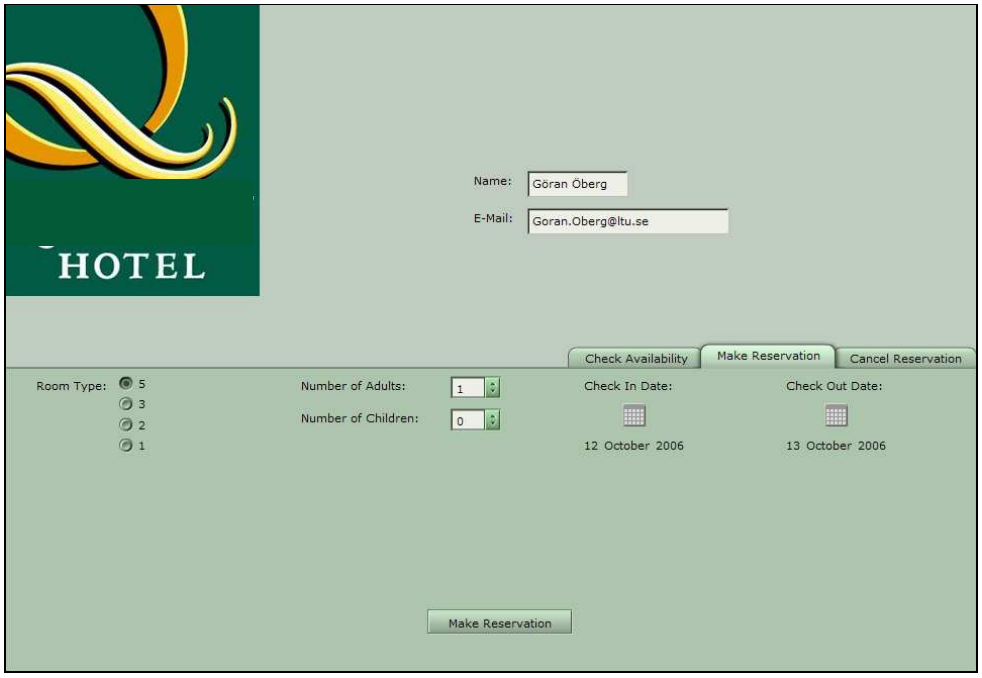


Figure 22 Original Example, Hotel reservation, "Make Reservation"

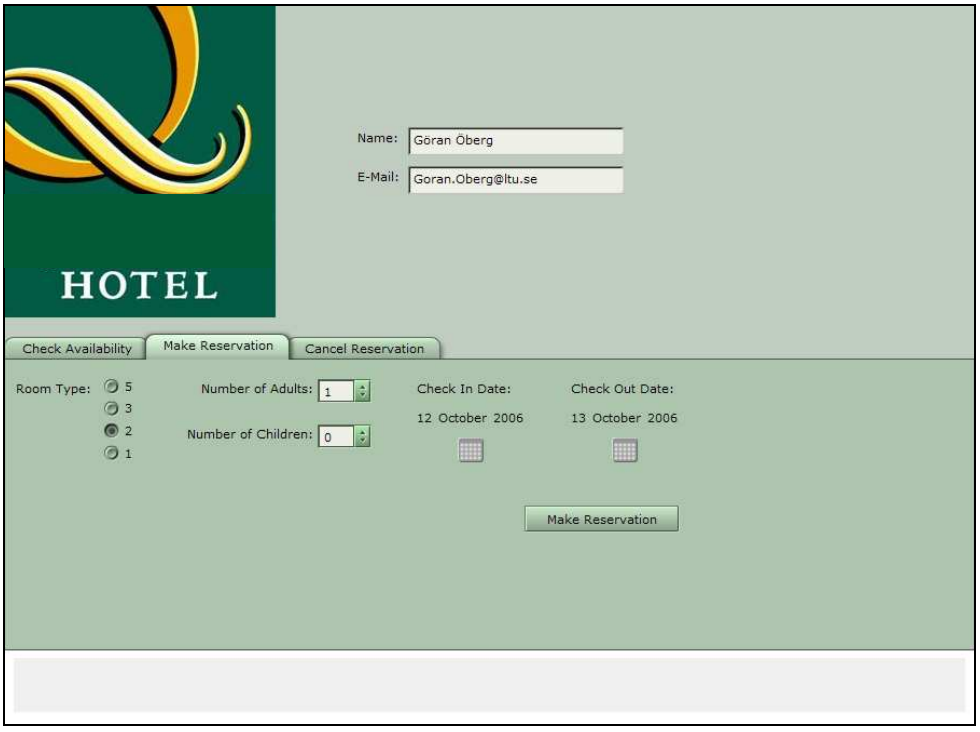


Figure 23 Updated example of "Make Reservation"

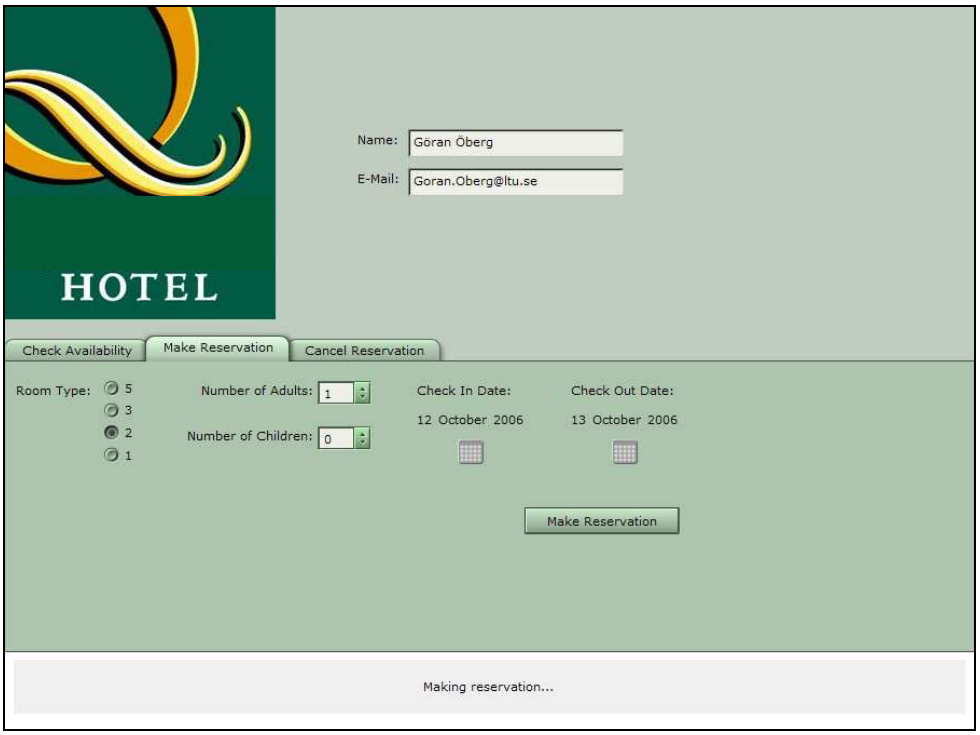


Figure 24 Updated example of "Make Reservation", reservation in progress

The task of making a reservation is almost identical in design, layout and data fields as the first task (5.2.3.1 Check Availability). The same comments apply for this action.

5.2.3.3. Cancel Reservation

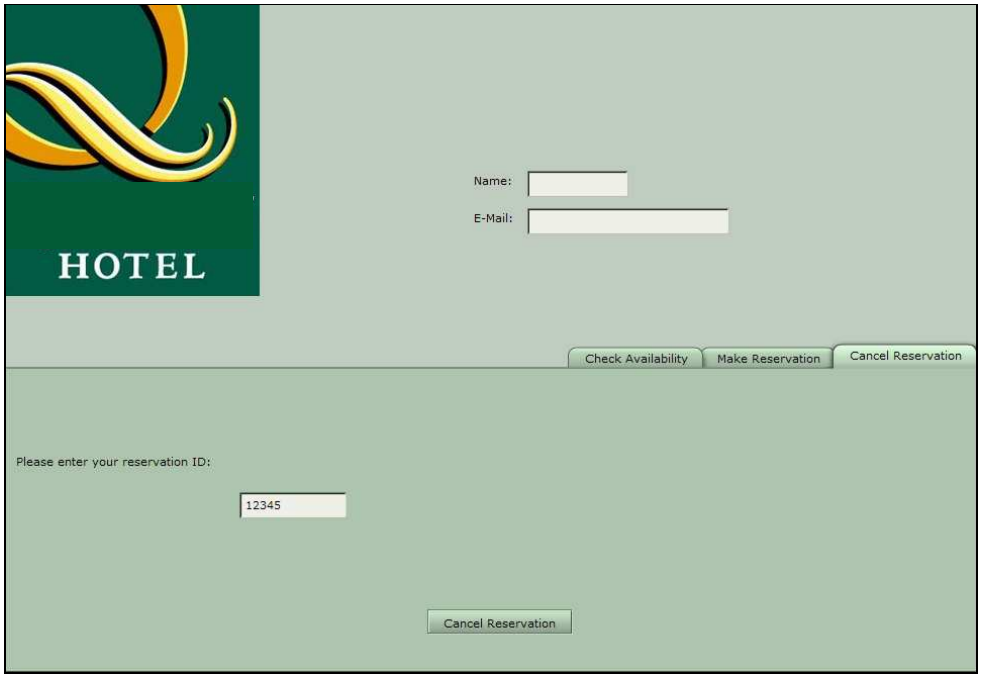


Figure 25 Original example of "Cancel Reservation"

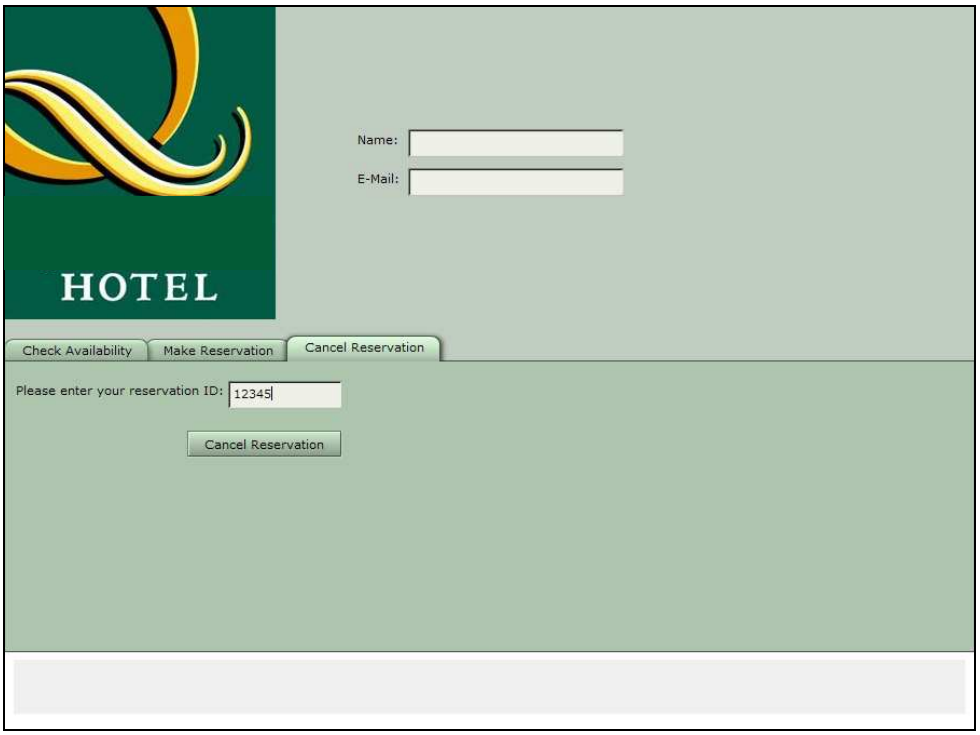


Figure 26 Updated example of "Cancel Reservation"

In this example it becomes very clear that the relocation of the tabs, input fields and activation button gives a user interface that is easier to overview. The place where information is expected to be entered is close to the text explaining what to enter and the subsequent task of sending the information with the “Cancel reservation” button is immediately underneath, also reminding the user of what action the information entered will be used for.

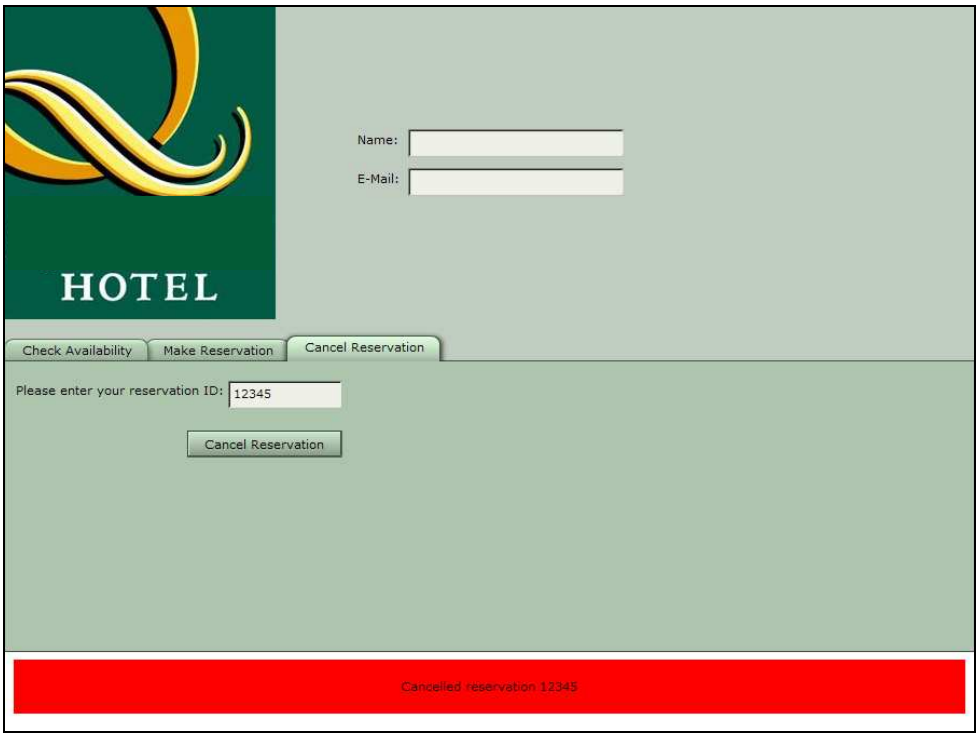


Figure 27 Updated example of "Cancel Reservation", cancellation succeded

The cancellation of an existing reservation may be considered of such importance that a more distinct alert of this action being completed may be warranted. ***Figure 27*** shows an example of how this could be presented. Preferably this alert message is visible only until the user begins a new task so as to not draw the user attentions from the next task.

5.2.4. Evaluation

The Hotel Reservation user interface example was improved by the use of the described user interface design patterns even though the improvements are mostly are in details.

The original user interface example was well planned and laid out in a good way giving a good overview of the tasks and what they consisted of. The rather few and well defined tasks of the service also lending themselves quite well to a straightforward user interface divided in distinct pages or tasks.

One of the remaining flaws of the user interface prototype is the selection of dates. This is mainly because the *datepicker* component of OpenLaszlo is broken in the way it reacts to configuration and interaction.

5.3. PC Wholesaler Example

The PC Wholesaler example user interface consists of mainly three tasks, browsing products, selecting products for ordering and placing the order.

5.3.1. Applied User Interface Design Patterns

The general changes to the user interface when applying user interface design patterns were:

- *Tree table*. The hierarchical structure of product categories and products lends itself very well to display using a tree. The categories expand to show the products within. The products are shown with their names and selecting them activates the right panel to show more info on the product as well as the possibility to order the product.
- *Two Panel Selectors*. The user interface is divided into a tree to the left showing the product categories and product names and a panel to the right showing information about the selected product. This also shows products selected for ordering and finally displays the forms for ordering and the messages about placed orders. The right panel also starts out with a presentation of the company and its services. This information can be recalled using the logotype at the top of the user interface, as is common on most web sites.
- *Same page messages*. The user interface design pattern originally called "*Same page error messages*" have been generalized to include all types of messages and is included using a separate component.
- *Diagonal balance*. The diagonal balance is not as distinct as in the Hotel Reservation example but is supported by the placement of activation buttons in the lower right. The dynamic nature of the hierarchical list is breaks the diagonal balance somewhat as its visual balance may shift during its use but this is a reasonable tradeoff between functionality and design.
- *Right aligned labels*. The labels have been aligned close to the input fields for clarity and minimizes the load on the user when making decision on what information to put where.

The user interface design patterns that were already present in the PC Wholesaler example user interface were:

- *Single line text field*. The single line text field is obviously suitable for short strings of text and should be preferably be wide enough so that the entered text is visible in its entirety all the time. Having the field show only part of what is entered is forcing the user to allocate more attention to the input and verification of the information and increases the risk for errors.
- *Prominent "done" button*. This button activates the form and is placed in the lower right to be found easily and are slightly oversized for clarity.

5.3.2. Other Changes

Other changes that were made, not matching a specific user interface design pattern:

- *Logo link*. The logotype at the top of the user interface is made into a link that returns the user to the presentation text that is first shown when at the start. This is a convention that is common on most web sites.
- *Minor errors eliminated*. In the previous design with a separate list of categories and products the user was able to corrupt the list of products by repeatedly selecting the text “Product Categories”. It was unclear why this was possible as there was code in place that seemed to clear the list of previous entries before adding new ones.

5.3.3. UI Tasks

5.3.3.1. Welcome screen

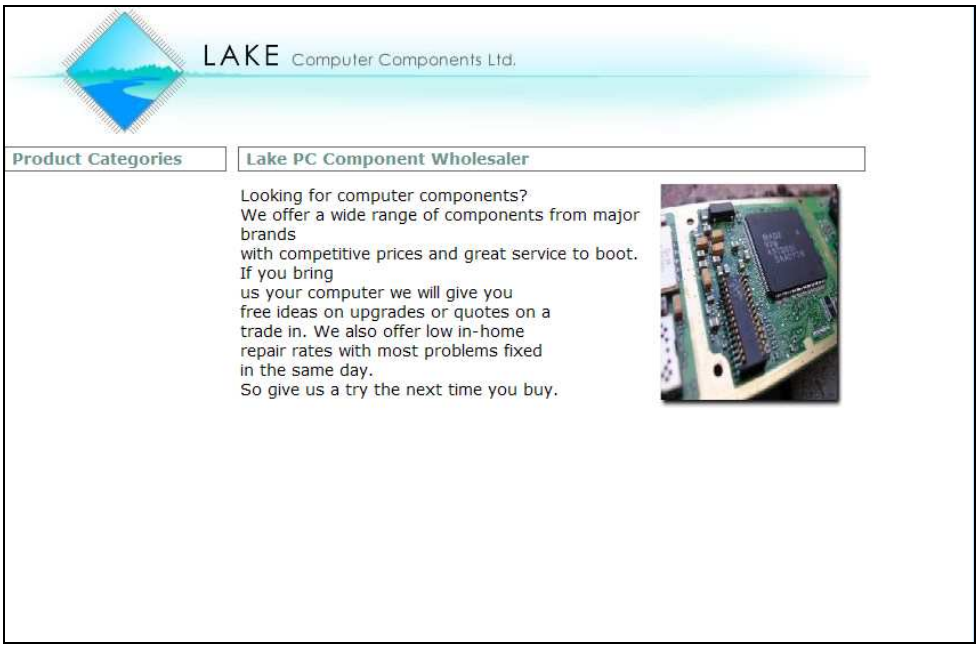


Figure 28 Original example, PC Wholesaler, welcome screen

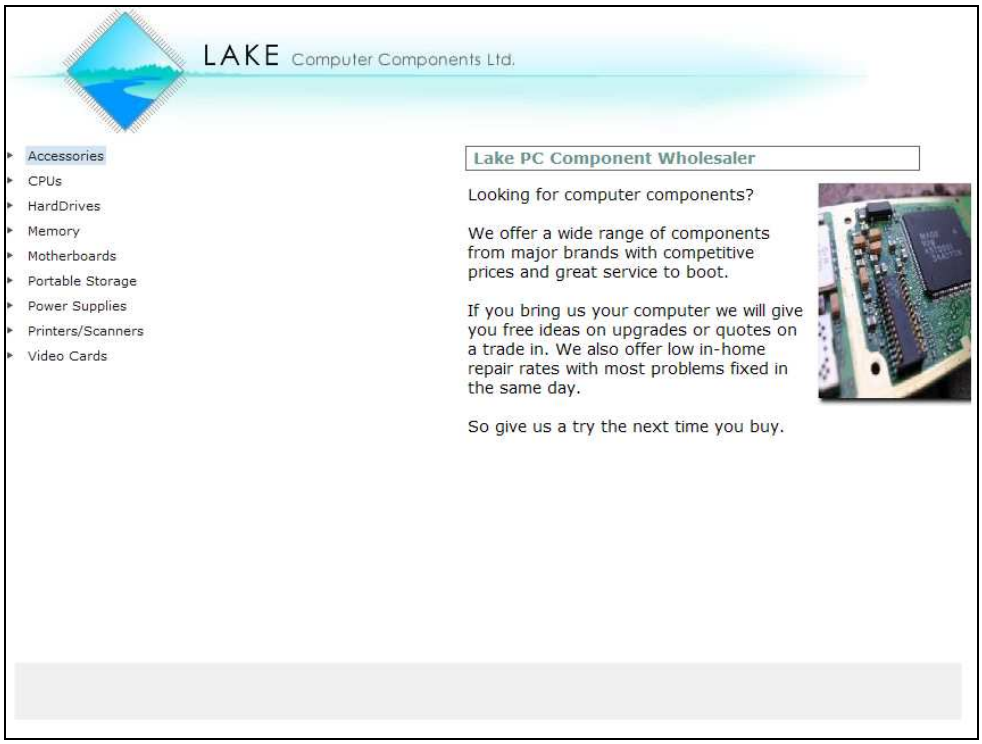


Figure 29 Updated example, PC Wholesaler, welcome screen

In both the original and updated example user interfaces, the first view shows a presentation of the company and its services. In the upgraded version this presentation can be reached by clicking on the company logo at the top.

5.3.3.2. Searching products

Figure 30 Original example, PC Wholesaler, showing product categories

Figure 31 Original example, PC Wholesaler, showing products

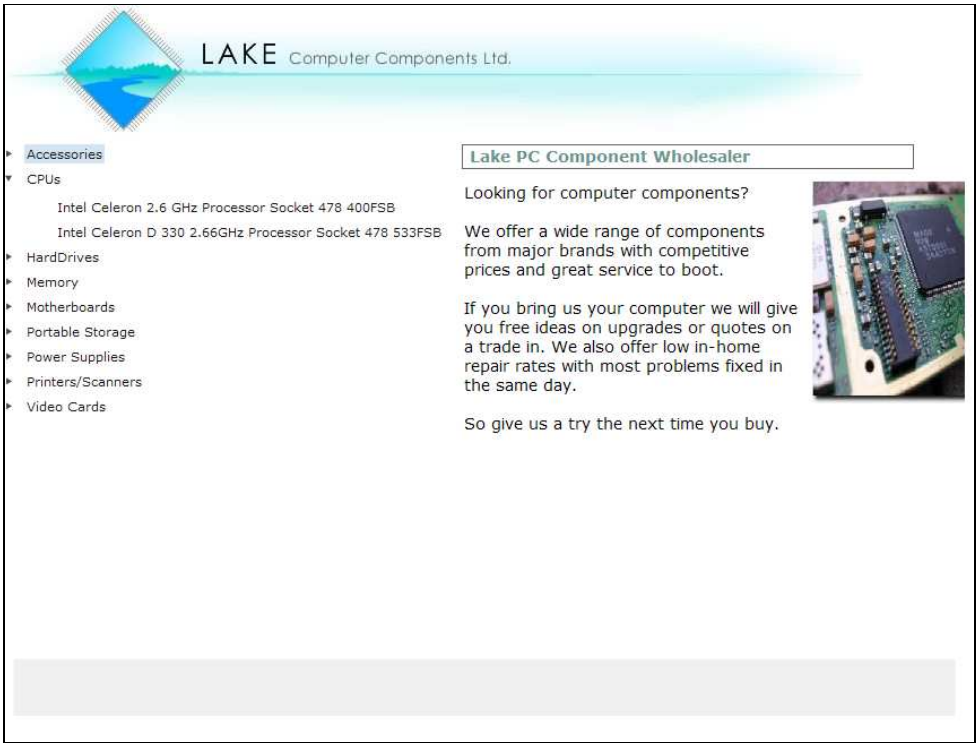


Figure 32 Updated example, PC Wholesaler, showing product categories and products

Searching for products with the original example requires the user to first select “Product Categories” to bring up the list of available product categories. The list of product categories can then be used to bring another list of available products in that category in the middle panel to the right of the product category list. The initial presentation is removed from the display when a product category is selected. This approach is largely modelled after the user interface design pattern *Cascading lists*.

The upgraded version has replaced the use of cascading lists with a tree view. This means that the user interface can be divided into two panels which have more stable placement and use. The tree view to the left expands into a tree and the panel has a scrollbar. This lets the user scroll up and down if the tree view is expanded to a larger height than the user interface allows to be shown at the same time.

5.3.3.3. Showing product information

Figure 33 Original example, PC Wholesaler, showing product information

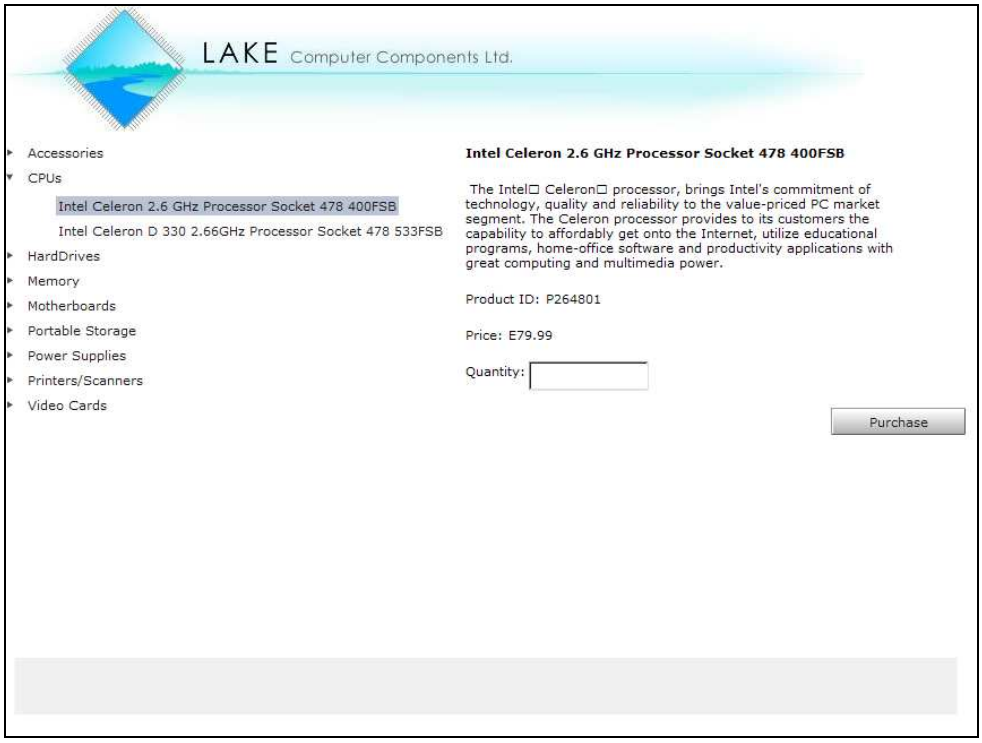


Figure 34 Updated example, PC Wholesaler, showing product information

Showing the information of a product shows the differences between the different versions clearly. The original uses three panels while the upgraded version is using two.

Some smaller differences in layout are the alignment of labels and input fields, different use of labels in front of product data and the use of the product name as a title for the presentation.

5.3.3.4. Adding product to order

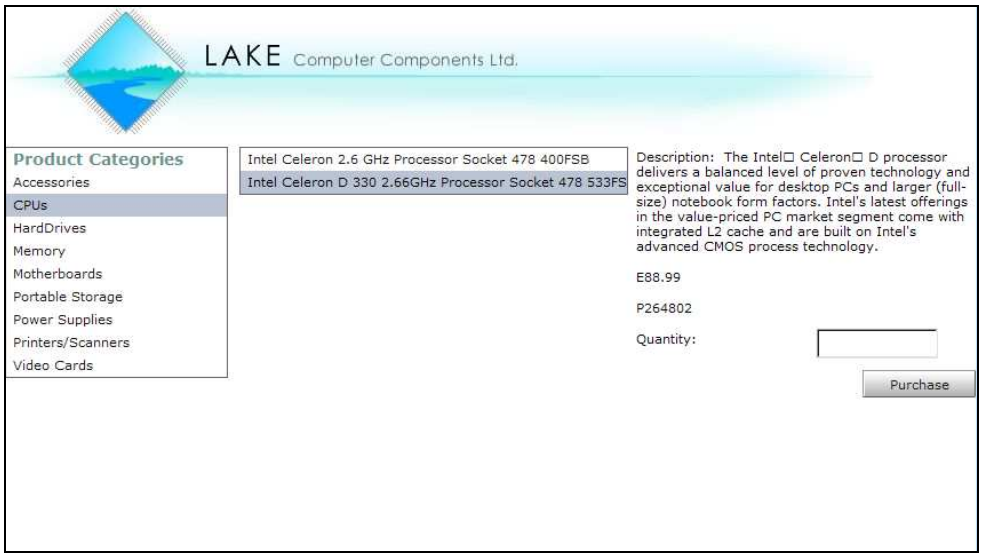


Figure 35 Original example, PC Wholesaler, adding item to order

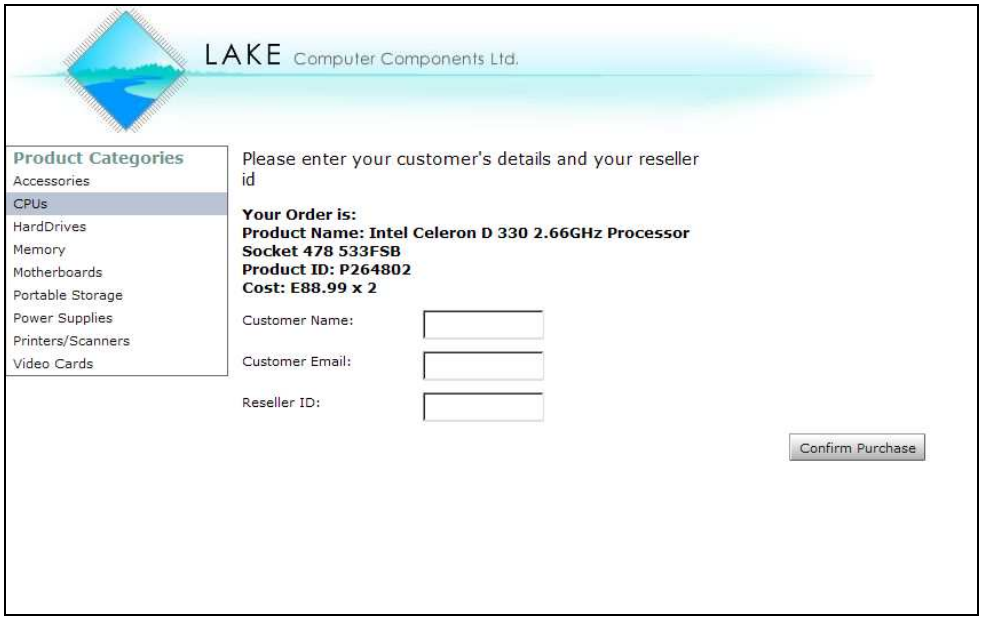


Figure 36 Original example, PC Wholesaler, item added to order

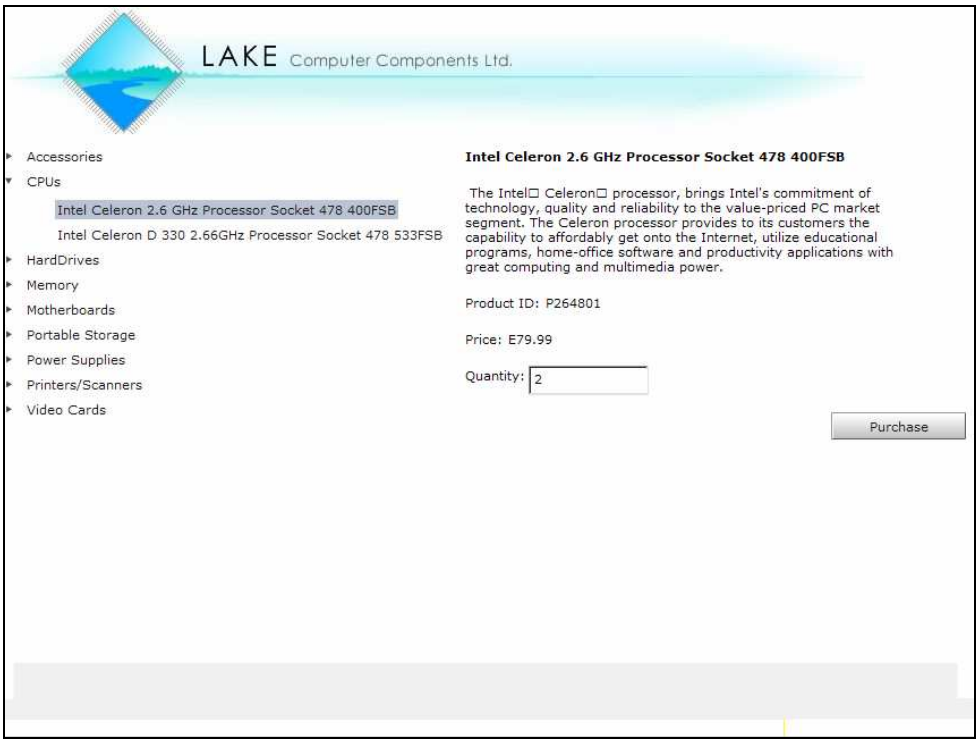


Figure 37 Updated example, PC Wholesaler, adding item to order

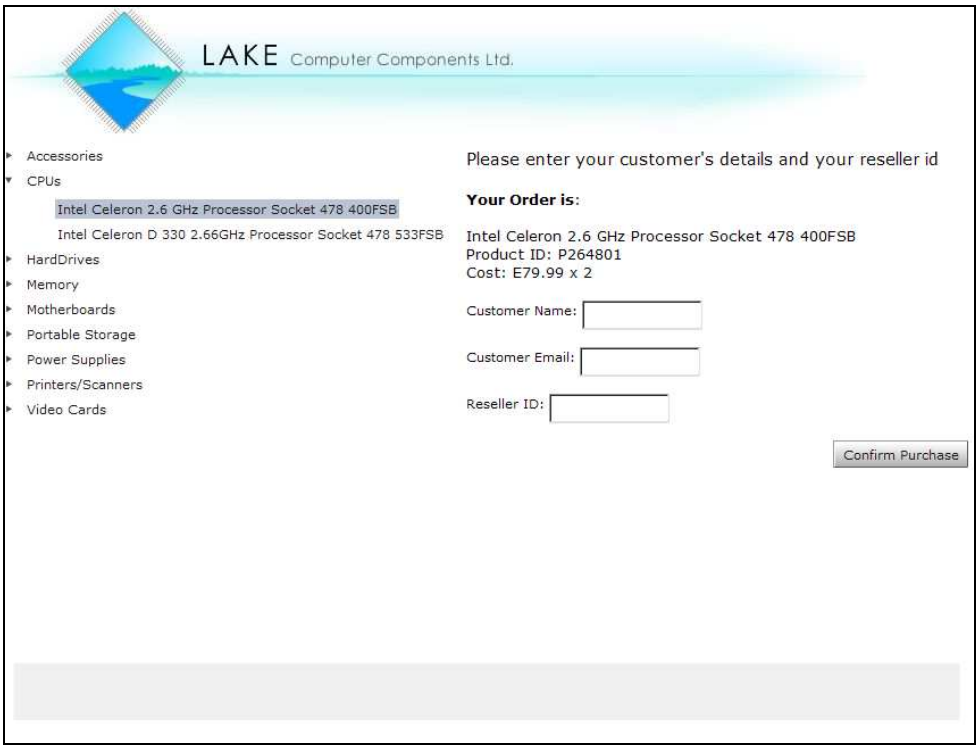


Figure 38 Updated example, PC Wholesaler, item added to order

To add an item to the order the product information view has an input field for quantities and a purchase button. The upgraded version has limited the valid input to the field to digits.

When an item is added, the user is shown the current list of items that have been added to the order list. The user can either continue to select further items to add or enter name, email and reseller id to confirm the purchase of the listed items.

One difference is that when the user adds an item to the list of items to order the panel previously showing the list of products is replaced by the current list of items. To continue adding products from the same product category the user must reselect the category. In the upgraded example the tree view of product categories and products remains the same when adding an item to the order list.

The order list can, in both the original and upgraded example, grow beyond the boundaries of the user interface thus making it useless as the button and input fields may become unreachable. The upgraded example was intended to solve this problem by adding a scrollbar that lets the user scroll the panel vertically whenever it gets too high for the available space. This did not work though as the layout of the components surrounding the order list and the scrollbar wouldn't align correctly. This is probably due to text added dynamically to the text area using program code and not having a fixed size from the start.

5.3.3.5. Placing order

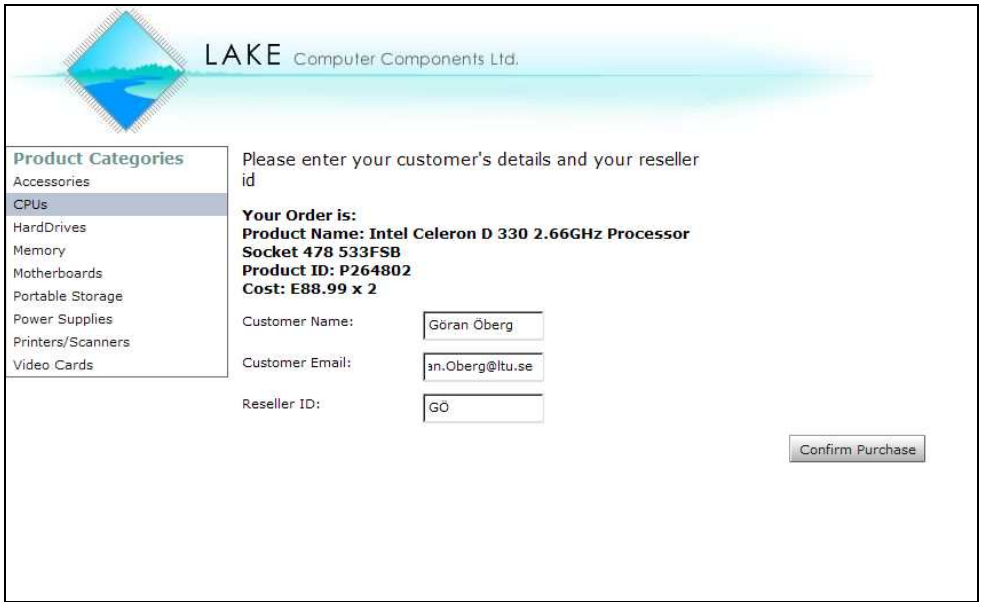


Figure 39 Original example, PC Wholesaler, placing order

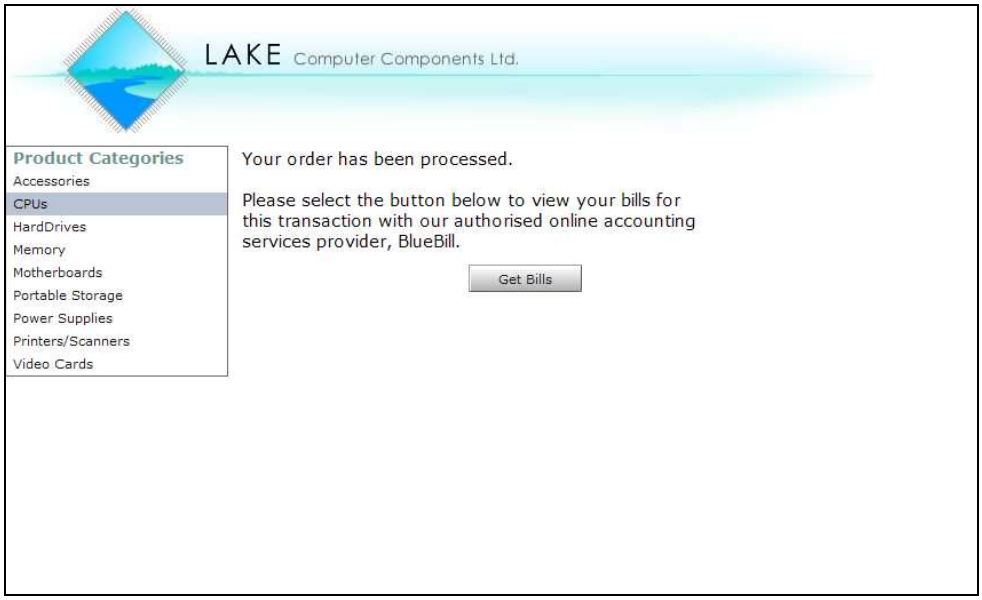


Figure 40 Original example, PC Wholesaler, order placed

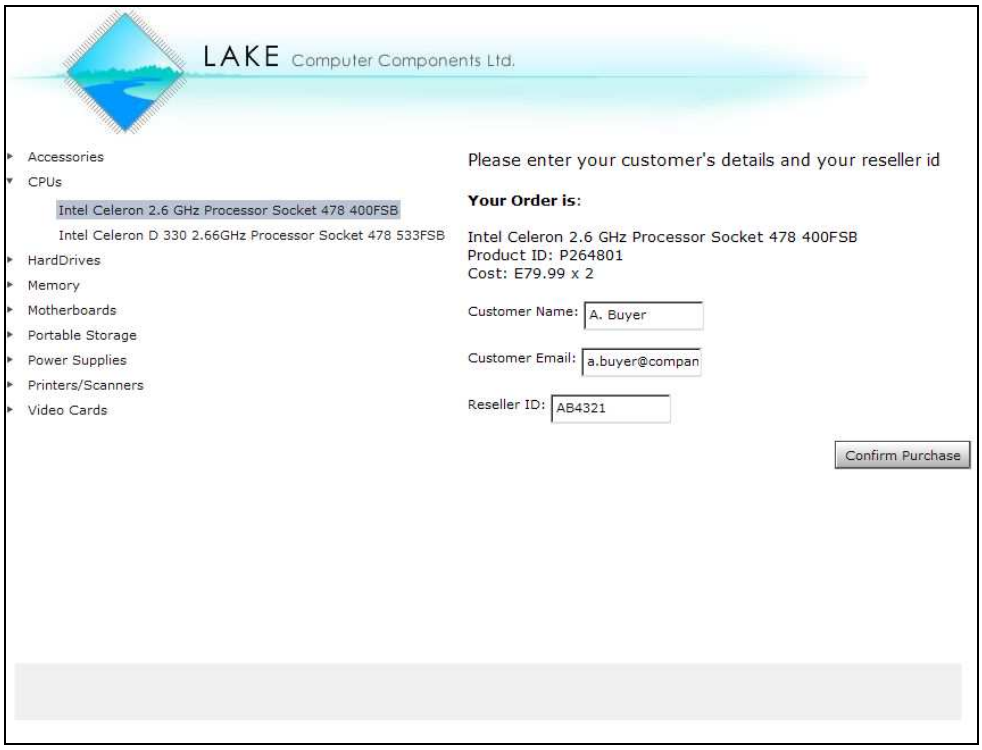


Figure 41 Updated example, PC Wholesaler, placing order

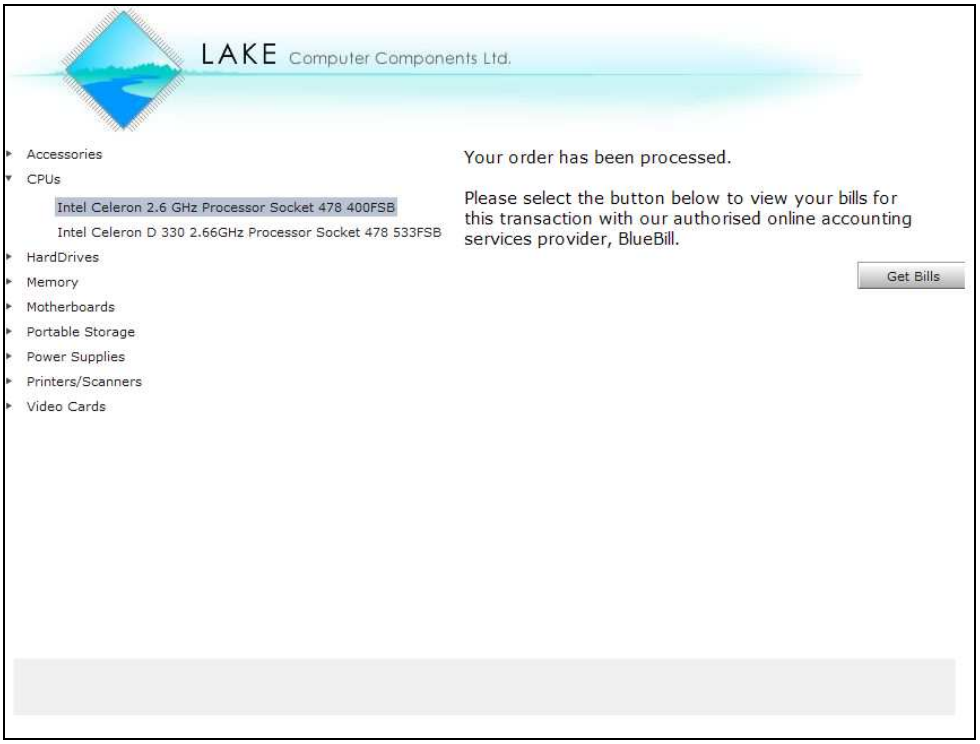


Figure 42 Updated example, PC Wholesaler, order placed

The actions of the original user interface example and the upgraded are similar when placing an order and also confirming the list of items that have been selected.

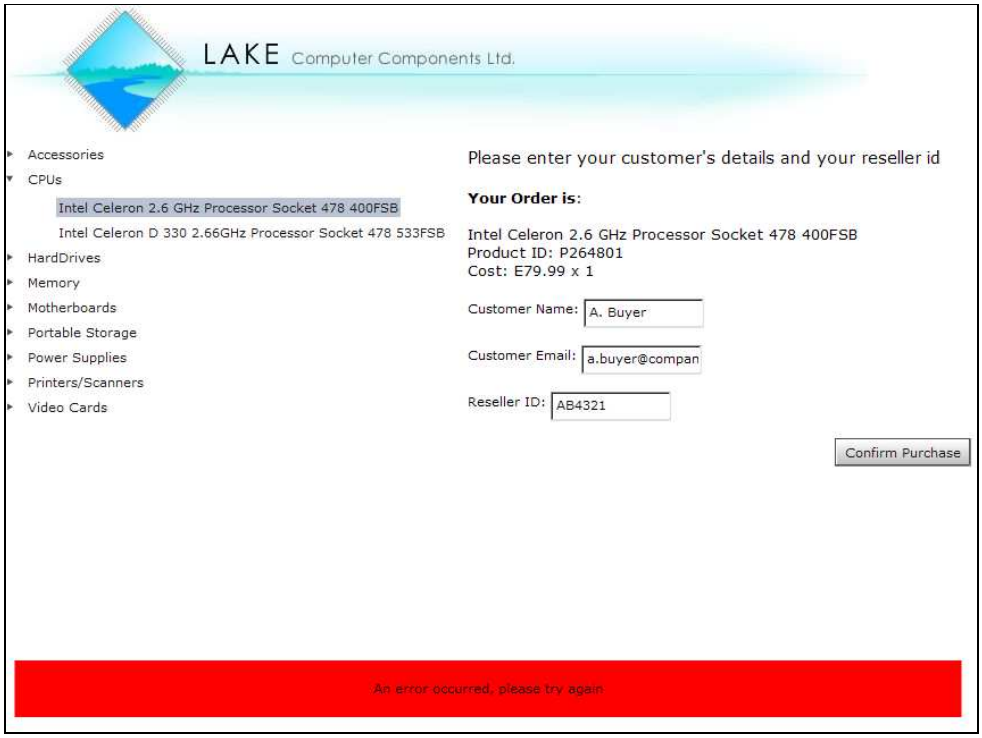


Figure 43 Updated example, PC Wholesaler, placing order, error occurred

When confirming the purchase there may be a communication error or perhaps an internal service . This is a good example of when the message area should be highlighted with a text informing the user that the confirmation of the purchase did not succeed.

5.3.4. Evaluation

The upgraded example user interface has a more consistent layout due to the use of a tree view and only two panels. The tree view also gives the user better overview and in some cases helps the user to continue the selection of items without having to reselect the product category.

Some problems have been encountered with OpenLaszlo, mostly connected with layout of dynamically sized objects and alignment.

5.4. Future Work

One problem with OpenLaszlo is how alignment works. In some instances it places objects in a very erratic way which have consequences for what is possible without resorting to a completely statically laid out interface.

A component library to facilitate the use of user interface design patterns would perhaps include components that alleviate these problems giving access to simple alignment and layout methods by working around the bugs in OpenLaszlo. The problems identified in OpenLaszlo will probably be solved but until then those problems can be obstacles to implementing the desired design of the user interface.

6. Implementation of UI Generator

6.1. Introduction

Within this deliverable an automatic user interface generator Eclipse plug-in has been developed for the DBE Studio. The plug-in provides a wizard for the generation of user interface prototypes from Service Description Language (SDL) models. SDL models are created using the DBE Studio's SDL editor. They can also be generated from the Business Modelling Language (BML) using the SSL2SDL generation plugin found in the DBE Studio.

6.2. Overview

The transformation from a SDL file to a user interface prototype is done from a wizard. The user selects some parameters that control the transformation of SDL using XSLT into an OpenLaszlo user interface. The prototype user interface is then opened in the OpenLaszlo editor, which is part of the DBE Studio. The UI generator plug-in is developed using Eclipse [63] and the Plug-in Development Environment (PDE).

6.3. Design and Implementation

The technical components that the DBE UI Generator Plug-in depends are the following:

6.3.1. DBE Studio

The DBE UI Generator is developed as a component in DBE Studio [62]. The SDL files that the UI Generator uses as inputs are modelled and created within DBE Studio. During development DBE Studio version 0.2.0 was used.

6.3.2. Eclipse

Eclipse [63] is the chosen framework for DBE Studio and therefore also for the development of the DBE UI Generator. The project started out using Eclipse version 3.1.2 but at the end of the project a switch to Eclipse 3.2 was made.

6.3.3. OpenLaszlo

The output of the DBE UI Generator is in the form of OpenLaszlo code. OpenLaszlo code is an implementation of XML where a user interface along with its logic is described in a simple object-oriented manner. When the generation of the user interface prototype is done, the wizard opens the resulting OpenLaszlo code in an OpenLaszlo editor provided by the OpenLaszlo IDE.

6.3.4. XSLT and XPath

The transformation from SDL to OpenLaszlo code is done using XSLT version 1.0 [64] and XPath version 1.0 [65]. The APIs used for accessing XSLT and XPath are found in the package javax.xml.transform in Java 2 Platform Standard Edition version 1.4.2.

The design of the UI generator is now outlined by the use of UML diagrams.

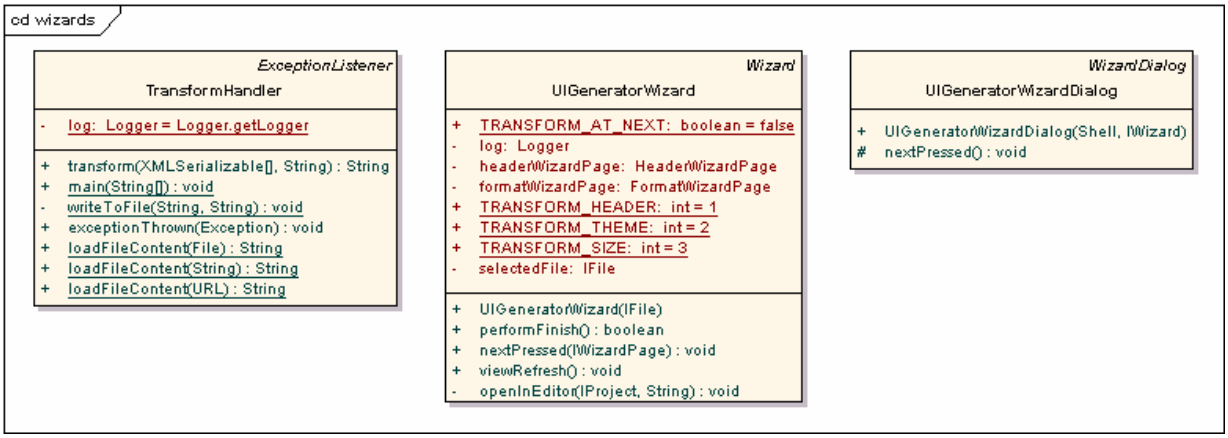


Figure 44 UML graph of package org.dbe.studio.tools.uigenerator.wizards

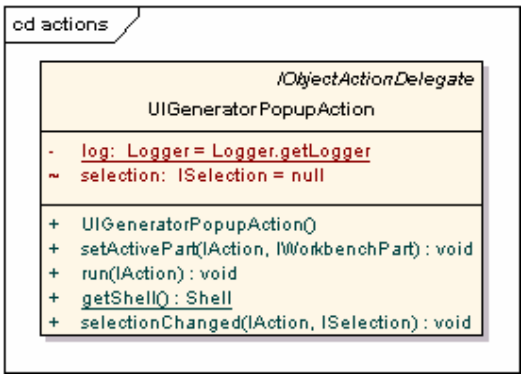


Figure 45 UML graph of package org.dbe.studio.tools.uigenerator.popup.actions

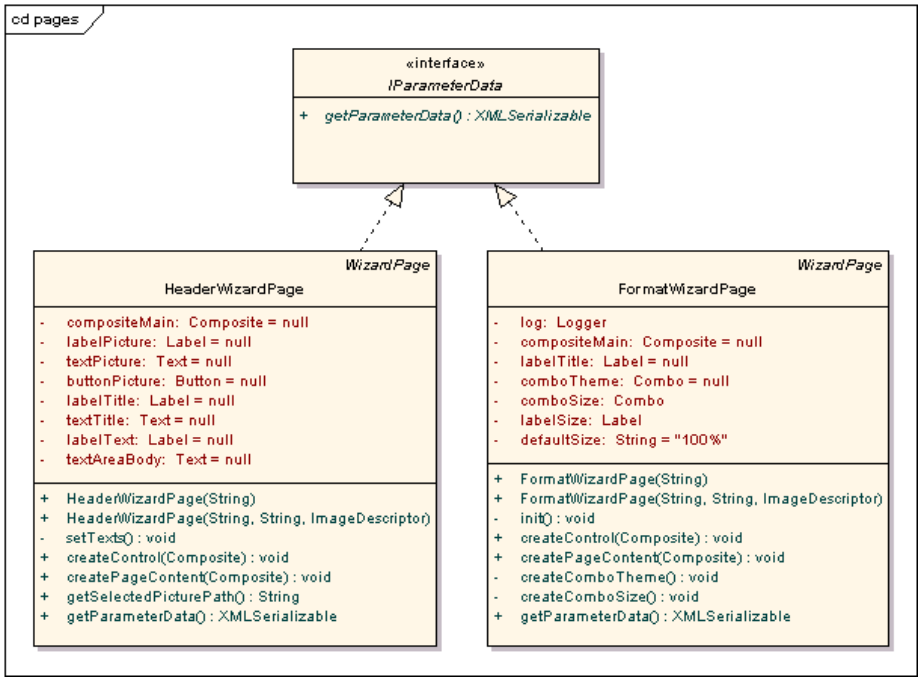


Figure 46 UML graph of package org.dbe.studio.tools.uigenerator.plugin.wizards.pages (partial)

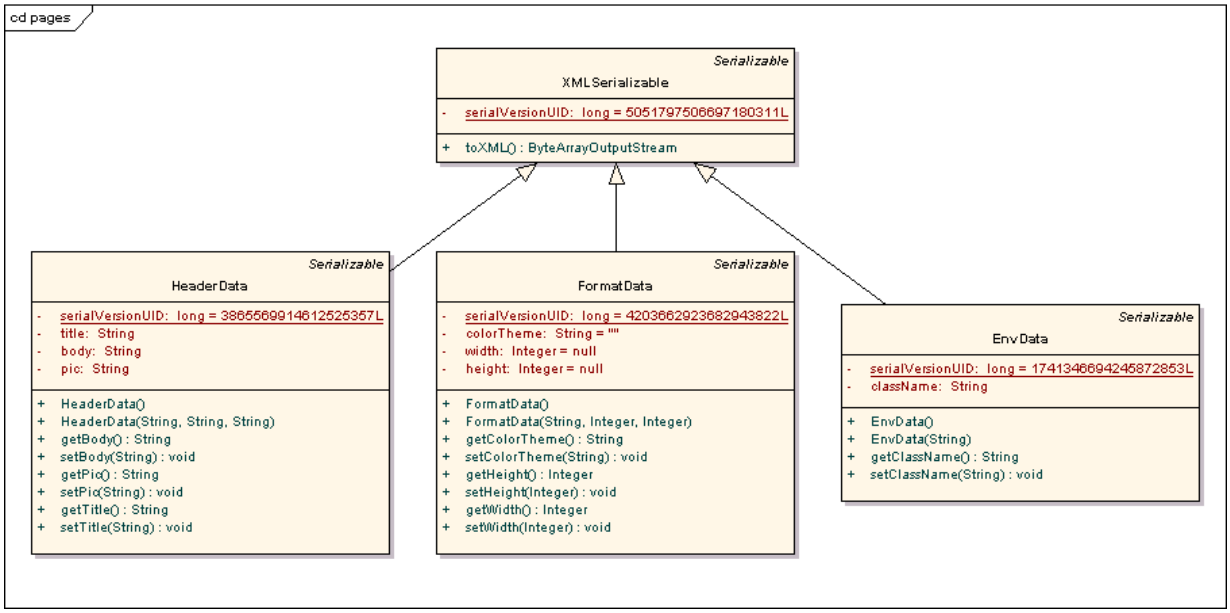


Figure 47 UML graph of package org.dbe.studio.tools.uigenerator.plugin.wizards.pages (partial)

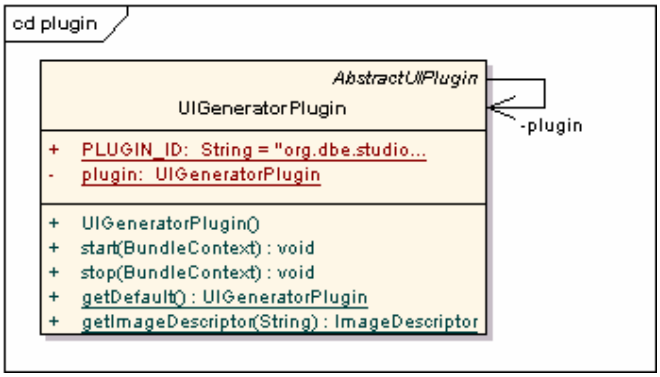


Figure 48 UML graph of package org.dbe.studio.tools.uigenerator.plugin

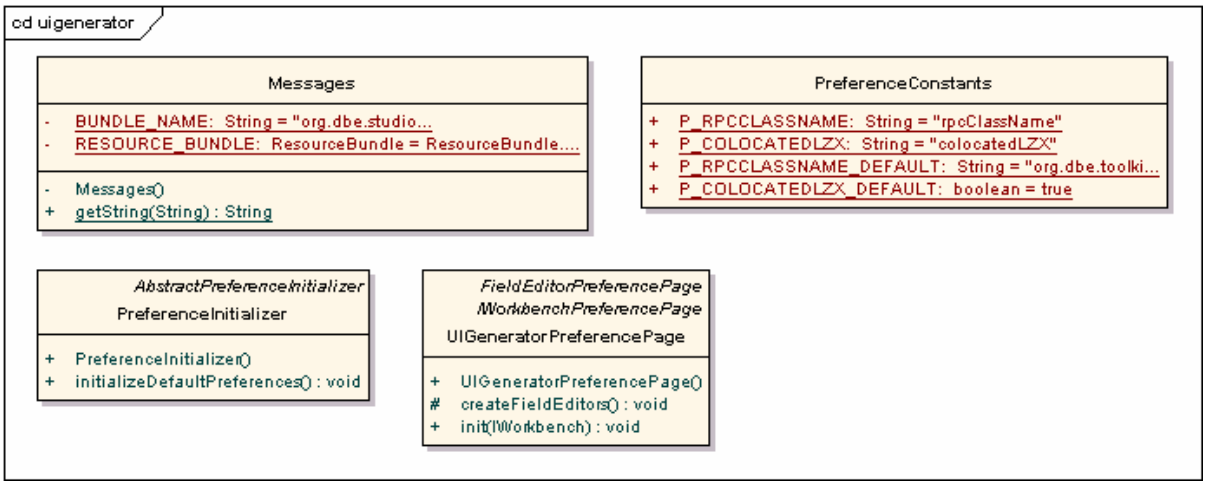


Figure 49 UML graph of package org.dbe.studio.tools.uigenerator

6.4. Internal dependencies

6.4.1. XSLT file general.xsl

The main transformation from SDL into OpenLaszlo code is the one contained in the file *general.xsl*. This transform has a number of dependencies:

- *The plug-in program code*. The passing of parameters from the Plugin wizard to the transform is made via java objects serialized in a XML format. The XML serialization describes the java object with its instance variables and their values. The SDL data and the serialized objects are combined in a simple XML concatenated format and passed to the transform. The parameters from the wizard are addressed from within the transform with XML entities (a construct similar to constants in most programming languages) identifying the classes.
- *The SDL schema*. The SDL format is described in the report *Deliverable 16.1 DBE Service description models and language definition* [32] [33] published by the DBE project. The transform is depending on naming of elements and the structure of the SDL document.

- *The OpenLaszlo library*. The XSLT transform code is tightly coupled with the OpenLaszlo standard library are used and how. This is also true for components from the DBE UI Design Pattern library.

6.4.2. DBE UI Generator Plug-in

The plug-in is dependent upon:

- *Parameters in externalized strings*. The plug-in are internationalized with all user interface messages and phrases separately stored in a properties file. Localization can be made to a large extent by translating the content of those properties.
- *Constants declared in PreferenceConstants.java*. The specifics of the preferences stored within Eclipse for the plug-in and the first-run defaults of those preferences are stored in PreferenceConstants.java.

6.4.3. OpenLaszlo Library

The OpenLaszlo library of components aimed at building DBE user interfaces have been developed using OpenLaszlo version 3.3.3. Currently the plug-in and its transform of a SDL file into a user interface prototype uses only one of those components, *dbe_panel.lzx*, where a specific area of the interface is reserved for displaying status messages and error alerts.

6.5. Plugin Usage

In order to use the DBE UI Generator plugin a user must have a valid SDL file in his/her DBE project. To invoke the UI Generator the user needs to right click on the SDL file and navigate to the entry as shown in ***Figure 50***.

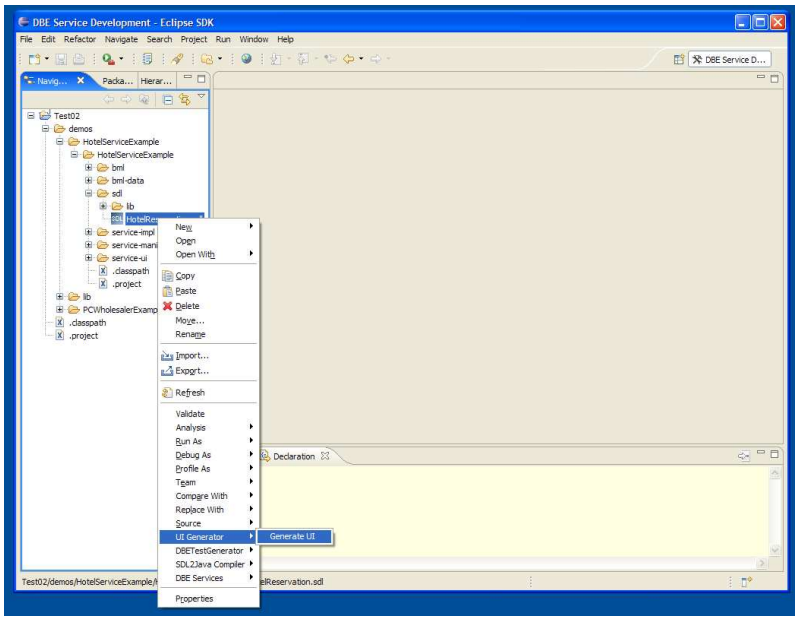


Figure 50 UI Generator Menu Entry

Once the user has selected the entry "Generate UI" the following wizard dialog will be presented (***Figure 51***).

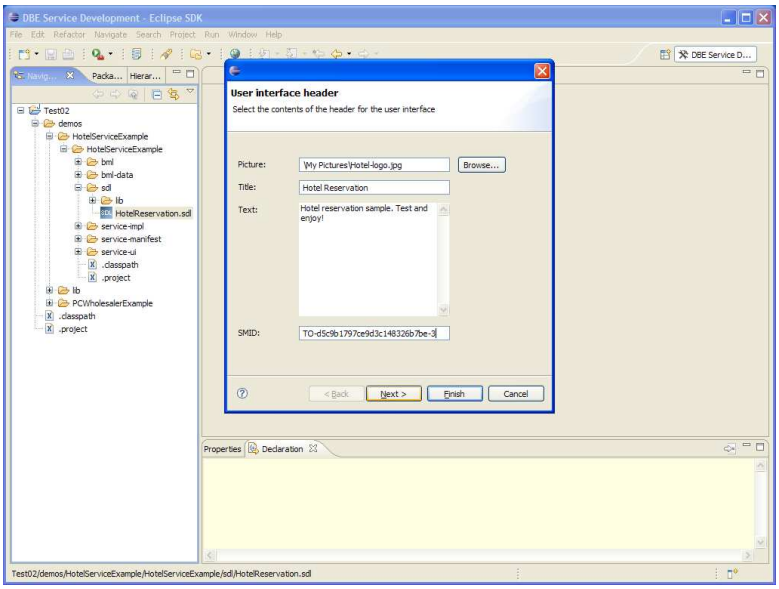


Figure 51 UI Generator 1st Wizard Page

In this dialog the user will be requested to enter in the required data and most importantly the Service Manifest Identifier (SMID) of the service which the generated UI is to communicate with. This requirement of providing the SMID manually will be removed as a maintenance update in a later version of the plugin.

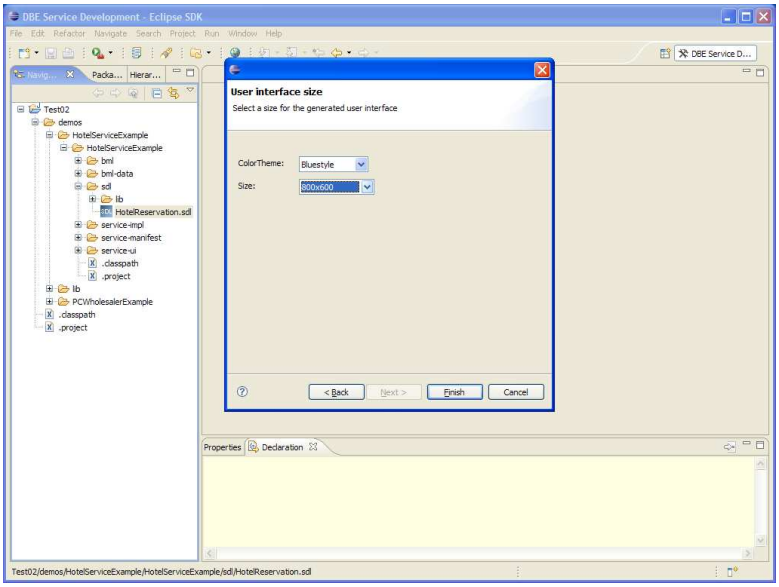


Figure 52 UI Generator 2nd Wizard Page

Once the required information is entered in the first wizard page, the user can then proceed on to the second wizard page (***Figure 52***). Here the user will be asked to specify the theme (visual styling) of the UI and also the size of the user interface view.

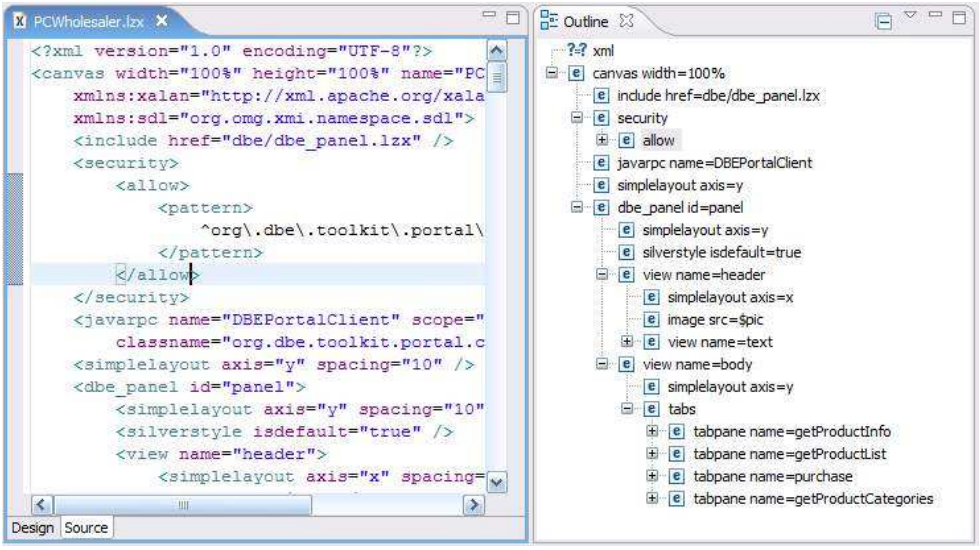


Figure 53 UI Generator Generated Code

Now with all the supplied information, the UI generator can then generate the OpenLaszlo code to represent the service as specified in the SDL. After generating the OpenLaszlo code, the UI generator then opens the code in the associated OpenLaszlo editor for further inspection and editing (***Figure 53***).

For the generated UI to be associated with a service, the developer needs to deploy the generated UI along with the developed service to a servent. This is done using the DBE Service Exporter. Once the UI and service have been exported to the servent the developer can expect to see a UI as shown in ***Figure 54***.

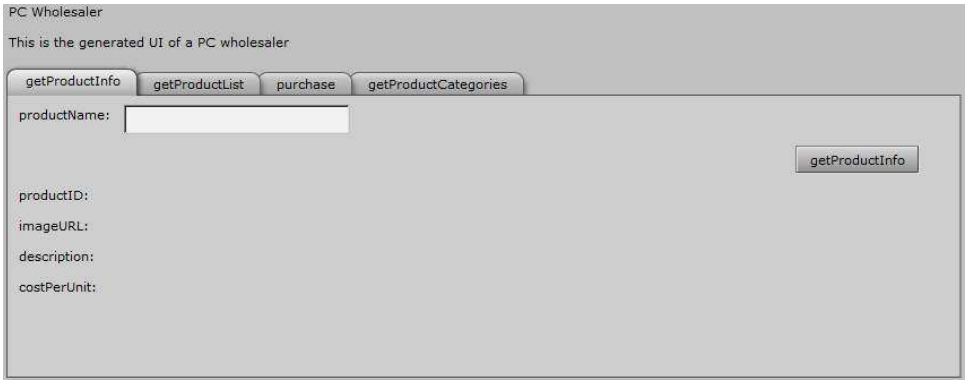


Figure 54 Example Generated User Interface

6.6. Future Work

During the development some areas of potential future work have been identified and will be described briefly:

- *Restoring edits after new conversions*: When a conversion of a SDL file have been done and the resulting user interface prototype edited, in the OpenLaszlo IDE or in some other way, these changes would be useful to re-apply to a new transform done from an updated SDL file.
- *Extended look-and-feel parameters*: The wizard needs to be extended with more options to control look-and-feel of the prototype user interface. This would help the user to produce consistent interfaces.
- *CSS support*: The wizard should be able to apply style sheets to the prototype user interface. This can be done either as a single theme style sheet covering general aspects of the design or with a set of style sheets, each providing control of specific features.
- *Iterative conversions*: When doing a conversion, especially if the wizard is extended with more parameters to control the conversion, an iterative workflow would be needed. The wizard would then let the user easily to step back in the wizard after reviewing the final conversion to alter parameters and regenerate the user interface prototype.
- *Intermediate previews*: Another effect if the wizard would be extended with several more parameters is that the user would benefit from having the ability to see a preview of what effect the parameters chosen so far will have. This could be either as a constantly updated generated preview or as a user requested one.
- *Per file preferences/selection memory*: To allow easily repeated generation of user interface prototypes from a specific SDL file, the wizard could store the set or parameters chosen for each file operated on.

- *Metadata modelling*: With the information currently given in the SDL file format only a rough prototype user interface is possible to build completely automatically. To be able to generate a functionally richer prototype the wizard would need more metadata about operations, data returned and their relations. The metadata needed could either be modelled as a separate task within this wizard or the modelling could be integrated in the tasks in other parts of DBE Studio.
- *Editing of resulting prototype*: The OpenLaszlo editor provided by the OpenLaszlo IDE has bugs and limitations that affect its use somewhat for the resulting user interface prototype. One such problem is that included images are not visible if the design view is used inline in the OpenLaszlo IDE (i.e. Local Design View settings). Another is the failure to produce a canvas with the intended size in the design view. Instead the interface is partly clipped with some parts not visible.

7. Conclusions

In this deliverable, the investigation began with a review of the state of the art in user interaction. This had an initial goal in mind to use design patterns as a language to transfer the findings of this report to DBE developers. The more specific goals as outlined in the introduction were:

1. Investigate the state of the art in user interaction in the context of user interfaces
2. Investigate UI methodologies for developing user interfaces, conveying them in a familiar fashion and demonstrate the application of those methodologies
3. Investigate tools to enable developer productivity when implementing user interfaces

Throughout the whole deliverable there has been an emphasis on reusability and this has been conveyed to developers in the form of a familiar mechanism, design patterns.

Design patterns are a powerful way to abstract the characteristics of design choices and design solutions. They can be used as a tool to allow more of those involved discuss the design process and the decisions to be made. They can help keep the focus on the general and higher-level aspects of the design.

One important aspect of design patterns is how they can form a design pattern language, describing a whole domain of design. This gives those involved in the design process the ability to describe and discuss problems and design choices in a more efficient and precise manner.

When developing or choosing a design pattern language the choice must also be made on the principles of how the design patterns are described. Several variants on the original design pattern descriptions found in Alexander's seminal work *A pattern language : towns, buildings, construction* [2] have been developed.

The variants can differ in a number if ways. The aspects and characteristics of each design patterns that are described can differ. Also, the way the different design patterns are categorized and grouped can differ. The abstraction level of what is described can also differ and may span wide and different areas of the domain. For example, some user interface design pattern languages focus on the visible components of a user interface (e.g. Tidwell[47, 48]) while others look at design patterns from the perspective of not only user-interface elements but also tasks and users.

Some of those variants on how to describe and develop design patterns and design pattern languages have been the source of some controversy. For example, the greatly appreciated book *Design Patterns – Elements of Reusable Object-Oriented Software*[16], which initiated the introduction of design patterns in the area of software engineering, has been criticized for the lack of support for user participation in the design process, something that was central in Alexander's original work.

In connection to this, some have claimed that the participatory element in design patterns is more fitting in user interface design patterns than in software engineering design patterns. This is because of the more tangible characteristics of user interfaces compared to program logics and software design.

Important aspects that may need to be addressed in further development of user interface design patterns are the temporal aspects. This is only very briefly addressed in the original work by Alexander and Borcher[5]. This work claims these aspects should be included in a richer way in user interface design pattern languages. This according to the authors would make them more successful in the HCI/user interface domain.

One conclusion for the further development and use of user interface design patterns is that the participatory aspects of design patterns need to be included and evaluated in the process.

The effectiveness and applicability of ready-to-use user interface design pattern libraries have been questioned as design patterns often are described with the ambition to capture the tension between proposed solution and the forces that promote and demote the use of it in specific situations. However, there are a number of projects underway that are developing user interface design patterns libraries aimed at specific areas (e.g. *Yahoo! Design Pattern Library*[56]). An important conclusion is that the development within such projects needs to be further studied to evaluate the applicability, limitations and suitable focus of such user interface design pattern libraries.

The components of the design pattern library, implemented as part of this task, were of somewhat limited use when existing user interfaces were to be upgraded. However, they are of more use when a user interface is built from scratch.

A conclusion drawn from this work is that design patterns in an implemented form place a demand on their level of flexibility. Indeed, the implementation of patterns may sometimes be of greater use as code examples and examples of the concepts of the user interface design patterns than completed and fixed components.

This aspect of user interface design patterns, not as fixed components in a graphical user interface but as patterns to be used for guidance when developing a user interface, might be especially significant when the development environment has the characteristics of OpenLaszlo, where the programming language in itself has a very high abstraction level. Many of the more basic user interface design patterns are already implemented and available as simple components.

It’s easy to underestimate the time it takes developing a user interface in OpenLaszlo, as layout behavior is sometime very surprising and layout flows that at first were thought to be easy to achieve. This was a motivating factor both in investigating the areas of computer assisted and automatic user interface generation and in implementing user interface generator for openlaszlo. It is hoped that with this generator, developer productivity could be accelerated.

From work carried out in this deliverable within the area of computer assisted user interface design, the research and development has, in many areas, just began. Specific domains of computer assisted user interface design have been explored and components of automation and extensive computer support have been found to be feasible.

There have not yet been any results that allow a general and highly automated user interface design process to take place. The design steps that can be automated often need the user to approve and adjust most of the design decisions for the process to be reliable. Completely automated user interface generation is not possible today other than in very specific domains. The conclusion from this is that until the area of computer assisted user interface design is better researched and developed, the computer assisted process is best used for rapid prototyping (e.g. generating skeleton user interfaces from higher level descriptions) and this rapid prototyping approach is reflected in the user interface generator.

8. References

1. Adobe Systems. Macromedia Flash content reaches 97.7% of Internet viewers, 2006, viewed 2006-08-14, <URL:http://www.adobe.com/products/player_census/flashplayer/>.
2. Alexander, C. *A pattern language : towns, buildings, construction / Christopher Alexander, Sara Ishikawa, Murray Silverstein with*, New York, 1977.
3. Alexander, C. *The timeless way of building*. Oxford University Press, New York, 1979.
4. Benyon, D., Turner, P. and Turner, S. *Designing interactive systems*. Addison-Wesley New York, 2005.
5. Borchers, J. Interaction design patterns: twelve theses. *position paper presented at the Workshop on Pattern Languages for Interaction Design, The CHI 2000 Conference on Human Factors in Computing Systems, April*. 2–3.
6. Borchers, J. *A pattern approach to interaction design*. Wiley, Chichester, England ; New York, 2001.
7. Bos, B., Lie, H.W., Lilley, C. and Jacobs, I. Cascading Style Sheets, level 2 CSS2 Specification. *W3C Recommendations are available at http://www.w3.org/TR/, May, 12*. 80.
8. Brewer, J. How people with disabilities use the Web, 2005, viewed 2006-08-14, <URL:http://www.w3.org/WAI/EO/Drafts/PWD-Use-Web/>.
9. Carr, D.A. A compact graphical representation of user interface interaction objects, research directed by Dept. of Computer Science.University of Maryland at College Park, 1995.
10. Dearden, A., Finlay, J., Allgar, E. and McManus, B. Using Pattern Languages in Participatory Design. *Proceedings of the Participatory Design Conference*.
11. Ecmascript, I.S.O. ISO/IEC 16262: 1998. *ECMAScript Language Specification*.
12. Eisenstein, J. and Puerta, A. Adaptation in automated user-interface design. *Proceedings of the 5th international conference on Intelligent user interfaces*. 74-81.
13. Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and Berners-Lee, T. Hypertext Transfer Protocol–HTTP/1.1, RFC 2616, June 1999, 1999.
14. Florins, M., Simarro, F.M., Vanderdonckt, J. and Michotte, B. Splitting rules for graceful degradation of user interfaces. *Proceedings of the working conference on Advanced visual interfaces*. 59-66.
15. Gajos, K. and Weld, D.S. SUPPLE: automatically generating user interfaces. *Proceedings of the 9th international conference on Intelligent user interface*. 93-100.

16. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

17. Gong, L. Java Security: Present AND Near Future.

18. Gosling, J., Bracha, G., Joy, B. and Steele, G.L. *The Java Language Specification*. Addison-Wesley Professional, 2000.

19. Graham, I. *A pattern language for Web usability*. Addison-Wesley, London ; Boston, 2003.

20. Hussey, A. Patterns for Safety and Usability in Human-Computer Interfaces. *Software Verification Research Centre TR99-05, The University of Queensland, January*.

21. Irons, M.L. Patterns for Personal Websites, 2003, viewed 2006-08-14, <URL:http://www.rdrop.com/~half/Creations/Writings/Web.patterns/>.

22. Iso, I.S.O. 8879-1986, Information processing-Text and Office Systems-Standard Generalized Markup Language (SGML). *International Organization for Standardization, October, 15*.

23. John, B. and Bergevin, H. Position is everything - Modern browser bugs explained in detail, 2006, viewed 2006-08-14, <URL:http://www.positioniseverything.net/>.

24. Koch, P. *ppk on JavaScript*. New Riders Press, 2006.

25. Krug, S. *Don't make me think! : a common sense approach to Web usability*. New Riders Pub., Berkeley, Calif, 2006.

26. Le Hors, A., Le Hegaret, P., Nicol, G., Robie, J., Champion, M. and Byrne, S. Document Object Model (DOM) Level 2 Core Specification Version 1.0. *W3C Recommendation, 13*.

27. Leveson, N.G. *Safeware: system safety and computers*. ACM Press New York, NY, USA, 1995.

28. Linda Dailey, P. Building Rich Web Applications with Ajax. *Computer, 38* (10). 14-17.

29. Lok, S., Feiner, S. and Ngai, G. Evaluation of visual balance for automated layout. *Proceedings of the 9th international conference on Intelligent user interface*. 101-108.

30. Mahemoff, M.J. and Johnston, L.J. Pattern Languages for Usability: An Investigation of Alternative Approaches. *Asia-Pacific Conference on Human Computer Interaction (APCHI), 98*. 25-31.

31. McComb, G. *ECMAScript language specification*. toExcel, San Jose, CA, 1999.

32. Montanari, G. D16.1: Service description models and language definition, Part 1 - SDL Definition, Soluta.net, 2005.

33. Montanari, G. D16.1: Service description models and language definition, Part 2 - Service Manifest Conceptual and Software Models, Soluta.net, 2005.

34. Nichols, J., Myers, B.A., Higgins, M., Hughes, J., Harris, T.K., Rosenfeld, R. and Pignol, M. Generating remote control interfaces for complex appliances. *Proceedings of the 15th annual ACM symposium on User interface software and technology*. 161-170.

35. O'Neill, E., Woodgate, D. and Kostakos, V. Easing the wait in the emergency room: building a theory of public information systems. *Proceedings of the 2004 conference on Designing interactive systems: processes, practices, methods, and techniques*. 17-25.

36. Pemberton, S. XHTML 1.0: The Extensible HyperText Markup Language. *World Wide Web Consortium Recommendation xhtml1, January*.

37. Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P. and Winograd, T. ICrafter: A Service Framework for Ubiquitous Computing Environments. *Proceedings of Ubicomp, 1*.

38. Puerta, A. and Eisenstein, J. XIML: A universal language for user interfaces, 2002, viewed 2006-08-14, <URL:http://www.ximl.org/documents/XimlWhitePaper.pdf>.

39. Raggett, D., Le Hors, A. and Jacobs, I. HTML 4.01 Specification. *W3C Recommendation REC-html401-19991224, World Wide Web Consortium (W3C), Dec*.

40. Reason, J. *Human error / James Reason*. Cambridge Univ. Press, Cambridge, 1990.

41. Refsnes Data. Browser Statistics, 2006, viewed 2006-08-14, <URL:http://www.w3schools.com/browsers/browsers_stats.asp>.

42. Rosenberg, D. Migrate apps from Internet Explorer to Mozilla, 2005, How to make Internet Explorer-specific Web applications work in Mozilla-based browsers, viewed 2006-08-14, <URL:http://www-128.ibm.com/developerworks/web/library/wa-ie2mozgd/>.

43. Shipman Iii, F.M. and McCall, R.J. Incremental formalization with the hyper-object substrate. *ACM Transactions on Information Systems (TOIS), 17* (2). 199-227.

44. Smith, K.C. Will the browser apply the rule(s)?, 2005, A comparison of support and behaviour of CSS in different web browsers, viewed 2006-08-14, <URL:http://centricle.com/ref/css/filters/>.

45. Sullivan, T. Debunking browser stats, 2005, viewed 2006-08-14, <URL:http://www.pantos.org/atw/f-35448.html>.

46. Tidwell, J. Common ground: A pattern language for human-computer interface design, 1990, viewed 2006-08-14, <URL:http://www.mit.edu/~jtidwell/interaction_patterns.html>.

47. Tidwell, J. *Designing Interfaces*. O'Reilly Media, Sebastopol, USA, 2005.

48. Tidwell, J. Designing Interfaces, 2005, viewed 2006-08-14, <URL:http://www.designinginterfaces.com/>.

49. Tidwell, J. The Gang of Four Are Guilty, 1999, viewed 2006-08-14, <URL:http://www.mit.edu/~jtidwell/gof_are_guilty.html>.

50. Van Duyne, D.K., Landay, J.A. and Hong, J.I. *The design of sites : patterns, principles, and processes for crafting a customer-centered Web experience*. Addison-Wesley, Boston, 2003.

51. Welie, M. Web Design Patterns, 2006, viewed 2006-09-14, <URL:http://www.welie.com/patterns/>.

52. World Web Wide Consortium. Document Object Model (DOM), 2005, viewed 2006-08-14, <URL:http://www.w3.org/DOM/>.

53. World Web Wide Consortium. HyperText Mark-up Language, 1992, Archive of one of the first drafts of the Hypertext Mark-up Language standard, viewed 2006-08-14, <URL:http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/MarkUp.html>.

54. World Web Wide Consortium. W3C (World Wide Web Consortium), viewed 2006-08-14, <URL:http://w3c.org/>.

55. World Web Wide Consortium. Web Accessibility Initiative (WAI), 2006, viewed 2006-08-14, <URL:http://www.w3.org/WAI/>.

56. Yahoo! Yahoo! Design pattern library, 2006, viewed 2006-08-14, <URL:http://developer.yahoo.com/ypatterns/>.

57. OpenLaszlo Framework and Runtime Engine, 2006, viewed 2006-11-12 <URL:http://www.openlaszlo.org>

58. Adobe Flash Framework and Authoring Environment, 2006, viewed 2006-11-12 <URL:http://www.adobe.com/products/flash>

59. World Web Wide Consortium. Dynamic Hyper Text Markup Language, 2006, 2006-11-12 <URL:http://www.w3c.org/DOM/faq.html#DHTML-DOM>

60. Formerly Macromedia Systems, now Adobe Systems Incorporated, 2006, viewed 2006-11-12 <URL:http://www.macromedia.com>

61. Adobe Systems Incorporated, 2006, viewed 2006-11-12 <URL:http://www.adobe.com>

62. DBE Studio - An integrated development for the DBE, 2006, viewed 2006-11-12 <URL:http://dbestudio.sourceforge.net>

63. Eclipse, An open development platform, 2006, viewed 2006-11-12 <URL:http://www.eclipse.org>

64. World Web Wide Consortium. XSL Transformations, 2006, viewed 2006-11-12 <URL:http://www.w3c.org/TR/xslt>

65. World Web Wide Consortium. XPath, 2006, viewed 2006-11-12

<URL: http://www.w3c.org/TR/xslt>

66. SUN Microsystems; M20.1: “Draft User Interface Specification”. Available from http://www.digital-ecosystem.org/.

9. Appendix

9.1. UI Generator JavaDoc documentation

Package Summary		Page
org.dbe.studio.tools.uigenerator	Provides for the creation of a preference page using field editors.	81
org.dbe.studio.tools.uigenerator.plugin	Provides for the connection to the Eclipse core, enabling the DBE UI Generator Wizard.	81
org.dbe.studio.tools.uigenerator.plugin.wizards	Provides for the generation of a wizard, including each wizard page as components.	81
org.dbe.studio.tools.uigenerator.plugin.wizards.pages	Provides two classes for each wizard page, one constructing the wizard GUI and one encapsulating its data.	81
org.dbe.studio.tools.uigenerator.popup.actions	Provides for the connection to the Eclipse content menu, invoking the DBE UI Generator Wizard.	81

Package org.dbe.studio.tools.uigenerator

Provides for the creation of a preference using field editors.

See:

Description

Class Summary		Page
Messages	Wrapper around ResourceBundle `ResourceBundle` for externalizing strings.	81
PreferenceConstants	Constant definitions for plug-in preferences	81
PreferenceInitializer	Initiates the UI Generator Plug-in for Eclipse preferences	81
UIGeneratorPreferencePage	Builds preference pages for the DBE UI Generator plugin using Field Editors.	81

Package org.dbe.studio.tools.uigenerator Description

Provides for the creation of a preference using field editors.

Class Messages

org.dbe.studio.tools.uigenerator

```
java.lang.Object
  └ org.dbe.studio.tools.uigenerator.Messages
```

---

public class **Messages**

extends Object

Wrapper around ResourceBundle `ResourceBundle` for externalizing strings.

Author:

Göran Öberg

See Also:

`ResourceBundle`

Method Summary		Page
static String	getString(String key) Resolves a key into an externalized string.	81

Method Detail

getString

```
public static String getString(String key)
```

Resolves a key into an externalized string.

Parameters:

`key` - key of externalized string

Returns:

externalized string

Class PreferenceConstants

org.dbe.studio.tools.uigenerator

java.lang.Object

└ **org.dbe.studio.tools.uigenerator.PreferenceConstants**

---

public class **PreferenceConstants**

extends Object

Constant definitions for plug-in preferences

Author:

Göran Öberg

Field Summary		Page
static final String	P_COLOCATEDLZX	81
static final boolean	P_COLOCATEDLZX_DEFAULT	81
static final String	P_RPCCLASSNAME	81
static final String	P_RPCCLASSNAME_DEFAULT	81

Constructor Summary	Page
PreferenceConstants()	81

Field Detail

P_RPCCLASSNAME

```
public static final String P_RPCCLASSNAME
```

P_COLOCATEDLZX

```
public static final String P_COLOCATEDLZX
```

P_RPCCLASSNAME_DEFAULT

```
public static final String P_RPCCLASSNAME_DEFAULT
```

P_COLOCATEDLZX_DEFAULT

```
public static final boolean P_COLOCATEDLZX_DEFAULT
```

Constructor Detail

PreferenceConstants

```
public PreferenceConstants()
```

Class PreferenceInitializer

org.dbe.studio.tools.uigenerator

```
java.lang.Object
  |
  org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer
    |
    org.dbe.studio.tools.uigenerator.PreferenceInitializer
```

public class **PreferenceInitializer**

extends org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer

Initiates the UI Generator Plug-in for Eclipse preferences

Author:

Göran Öberg

Constructor Summary	Page
PreferenceInitializer()	81

Method Summary		Page
void	initializeDefaultPreferences()	81

Methods inherited from class org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer
initializeDefaultPreferences

Constructor Detail

PreferenceInitializer

```
public PreferenceInitializer()
```

Method Detail

initializeDefaultPreferences

```
public void initializeDefaultPreferences()
```

Overrides:

initializeDefaultPreferences in class org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer

See Also:

org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer.initializeDefaultPreferences()

Class UIGeneratorPreferencePage

org.dbe.studio.tools.uigenerator

```
java.lang.Object
  └org.eclipse.jface.dialogs.DialogPage
      └org.eclipse.jface.preference.PreferencePage
          └org.eclipse.jface.preference.FieldEditorPreferencePage
              └org.dbe.studio.tools.uigenerator.UIGeneratorPreferencePage
```

All Implemented Interfaces:

EventListener, org.eclipse.jface.dialogs.IDialogPage, org.eclipse.jface.dialogs.IMessageProvider, org.eclipse.jface.preference.IPreferencePage, org.eclipse.jface.util.IPropertyChangeListener, org.eclipse.ui.IWorkbenchPreferencePage

---

public class **UIGeneratorPreferencePage**

extends org.eclipse.jface.preference.FieldEditorPreferencePage

implements org.eclipse.ui.IWorkbenchPreferencePage

Builds preference pages for the DBE UI Generator plugin using Field Editors.

Author:

Göran Öberg

Fields inherited from class org.eclipse.jface.preference.FieldEditorPreferencePage
FLAT, GRID

Fields inherited from interface org.eclipse.jface.dialogs.IMessageProvider
ERROR, INFORMATION, NONE, WARNING

Constructor Summary	Page
UIGeneratorPreferencePage() The default constructor.	81

Method Summary		Page
final void	**init**(org.eclipse.ui.IWorkbench workbench) Initializes the preference store.	81

Methods inherited from class org.eclipse.jface.preference.FieldEditorPreferencePage
dispose, performOk, propertyChange, setVisible

Methods inherited from class org.eclipse.jface.preference.PreferencePage
applyData, computeSize, createControl, getContainer, getPreferenceStore, isValid, okToLeave, performCancel, performHelp, performOk, setContainer, setErrorMessage, setMessage, setPreferenceStore, setSize, setTitle, setValid, toString

Methods inherited from class org.eclipse.jface.dialogs.DialogPage
dispose, getControl, getDescription, getErrorMessage, getImage, getMessage, getMessageType, getShell, getTitle, performHelp, setDescription, setErrorMessage, setImageDescriptor, setMessage, setMessage, setTitle, setVisible

Methods inherited from interface org.eclipse.jface.dialogs.IDialogPage
createControl, dispose, getControl, getDescription, getErrorMessage, getImage, getMessage, getTitle, performHelp, setDescription, setImageDescriptor, setTitle, setVisible

Methods inherited from interface org.eclipse.jface.dialogs.IMessageProvider
getMessage, getMessageType

Methods inherited from interface org.eclipse.jface.preference.IPreferencePage
computeSize, isValid, okToLeave, performCancel, performOk, setContainer, setSize

Methods inherited from interface org.eclipse.jface.util.IPropertyChangeListener
propertyChange

Methods inherited from interface org.eclipse.ui.IWorkbenchPreferencePage
init

Constructor Detail

UIGeneratorPreferencePage

```
public UIGeneratorPreferencePage()
```

The default constructor.

Method Detail

init

```
public final void init(org.eclipse.ui.IWorkbench workbench)
```

Initializes the preference store.

Specified by:

`init` in interface `org.eclipse.ui.IWorkbenchPreferencePage`

See Also:

`org.eclipse.ui.IWorkbenchPreferencePage.init(org.eclipse.ui.IWorkbench)`

Package org.dbe.studio.tools.uigenerator.plugin

Provides for the connection to the Eclipse core, enabling the DBE UI Generator Wizard.

See:

Description

Class Summary		Page
UIGeneratorPlugin	Main plugin class called from the Eclipse UI.	81

Package org.dbe.studio.tools.uigenerator.plugin Description

Provides for the connection to the Eclipse core, enabling the DBE UI Generator Wizard.

Class UIGeneratorPlugin

org.dbe.studio.tools.uigenerator.plugin

```
java.lang.Object
  └ org.eclipse.core.runtime.Plugin
      └ org.eclipse.ui.plugin.AbstractUIPlugin
          └ org.dbe.studio.tools.uigenerator.plugin.UIGeneratorPlugin
```

All Implemented Interfaces:

org.osgi.framework.BundleActivator

---

public class **UIGeneratorPlugin**

extends org.eclipse.ui.plugin.AbstractUIPlugin

Main plugin class called from the Eclipse UI.

Field Summary	Page
static final String **PLUGIN_ID**	81

Fields inherited from class org.eclipse.core.runtime.Plugin
PLUGIN_PREFERENCE_SCOPE, PREFERENCES_DEFAULT_OVERRIDE_BASE_NAME, PREFERENCES_DEFAULT_OVERRIDE_FILE_NAME

Constructor Summary	Page
UIGeneratorPlugin() Sole constructor.	81

Method Summary		Page
static UIGeneratorPlugin	**getDefault**() Returns the shared instance.	81

static org.eclipse.jface.resource.ImageDescriptor	**getImageDescriptor**(String path) Returns an image descriptor for the image file at the given plug-in relative path.	81
void	**start**(org.osgi.framework.BundleContext context) Called upon plug-in activation.	81
void	**stop**(org.osgi.framework.BundleContext context) Called when the plug-in is stopped	81

Methods inherited from class org.eclipse.ui.plugin.AbstractUIPlugin
getDialogSettings, getImageRegistry, getPreferenceStore, getWorkbench, imageDescriptorFromPlugin, shutdown, start, startup, stop

Methods inherited from class org.eclipse.core.runtime.Plugin
find, find, getBundle, getDescriptor, getLog, getPluginPreferences, getStateLocation, internalInitializeDefaultPluginPreferences, isDebugging, openStream, openStream, savePluginPreferences, setDebugging, shutdown, start, startup, stop, toString

Methods inherited from interface org.osgi.framework.BundleActivator
start, stop

Field Detail

PLUGIN_ID

```
public static final String PLUGIN_ID
```

Constructor Detail

UIGeneratorPlugin

```
public UIGeneratorPlugin()
```

Sole constructor.

Method Detail

start

```
public void start(org.osgi.framework.BundleContext context)
          throws Exception
```

Called upon plug-in activation.

Specified by:

`start` in interface `org.osgi.framework.BundleActivator`

Overrides:

`start` in class `org.eclipse.ui.plugin.AbstractUIPlugin`

Throws:

`Exception`

---

stop

```
public void stop(org.osgi.framework.BundleContext context)
         throws Exception
```

Called when the plug-in is stopped

Specified by:

`stop` in interface `org.osgi.framework.BundleActivator`

Overrides:

`stop` in class `org.eclipse.ui.plugin.AbstractUIPlugin`

Throws:

`Exception`

---

getDefault

```
public static UIGeneratorPlugin getDefault()
```

Returns the shared instance.

Returns:

plugin UIGeneratorPlugin

---

getImageDescriptor

```
public static org.eclipse.jface.resource.ImageDescriptor getImageDescriptor(String path)
```

Returns an image descriptor for the image file at the given plug-in relative path.

Parameters:

`path` - plug-in relative path to image file

Returns:

the image descriptor

Package org.dbe.studio.tools.uigenerator.plugin.wizards

Provides for the generation of a wizard, including each wizard page as components.

See:

Description

Class Summary		Page
TransformHandler	Provides transform tools for transforming SDL into OpenLaszlo code using XSLT.	81
UIGeneratorWizard	Implements a wizard with wizard pages from package org.dbe.studio.tools.uigenerator.plugin.wizards.pages.org.dbe.studio.tools.uigenerator.plugin.wizards.pages.	81
UIGeneratorWizardDialog	Constructs a wizard dialog.	81

Package org.dbe.studio.tools.uigenerator.plugin.wizards Description

Provides for the generation of a wizard, including each wizard page as components.

Class TransformHandler

org.dbe.studio.tools.uigenerator.plugin.wizards

```
java.lang.Object
  └ org.dbe.studio.tools.uigenerator.plugin.wizards.TransformHandler
```

All Implemented Interfaces:

ExceptionListener

---

public class **TransformHandler**

extends Object

implements ExceptionListener

Provides transform tools for transforming SDL into OpenLaszlo code using XSLT.

Author:

Göran Öberg

Constructor Summary	Page
TransformHandler()	81

Method Summary		Page
void	exceptionThrown(Exception e) Exceptions thrown when XMLEncoder parses object is handled and logged.	81
static String	loadFileContent(File file) Simple routine to load a file's content and return as string.	81
static String	loadFileContent(String filename) Simple utility to load a file's content and return as string.	81
static String	loadFileContent(URL url) Simple utility to load a file's content and return as string.	81
static void	main(String[] args) Debugging development method.	81

String	**transform**(XMLSerializable[] xmlser, String sdl)	81

Constructor Detail

TransformHandler

```
public TransformHandler()
```

Method Detail

transform

```
public String transform(XMLSerializable[] xmlser,
                        String sdl)
                 throws IOException
```

Throws:

IOException

---

main

```
public static void main(String[] args)
```

Debugging development method. Test of local code, doing conversions to and from test files.

Parameters:

args - any command line arguments, ignored

---

exceptionThrown

```
public void exceptionThrown(Exception e)
```

Exceptions thrown when XMLEncoder parses object is handled and logged.

Specified by:

exceptionThrown in interface ExceptionListener

Parameters:

e - exception to handle

loadFileContent

```
public static String loadFileContent(File file)
                             throws Exception
```

Simple routine to load a file's content and return as string.

Parameters:

`file` - file to load content from

Returns:

content of file

Throws:

`Exception` - if any file input error occurs.

---

loadFileContent

```
public static String loadFileContent(String filename)
                             throws Exception
```

Simple utility to load a file's content and return as string.

Parameters:

`filename` - filename to load content from

Returns:

content of file

Throws:

`Exception` - if any file input error occurs

---

loadFileContent

```
public static String loadFileContent(URL url)
                             throws Exception
```

Simple utility to load a file's content and return as string.

Parameters:

`url` - URL to load content from

Returns:

content of file

Throws:

`Exception` - if any file input error occurs

Class UIGeneratorWizard

org.dbe.studio.tools.uigenerator.plugin.wizards

```
java.lang.Object
  └ org.eclipse.jface.wizard.Wizard
      └ org.dbe.studio.tools.uigenerator.plugin.wizards.UIGeneratorWizard
```

All Implemented Interfaces:

org.eclipse.jface.wizard.IWizard

---

public class **UIGeneratorWizard**

extends org.eclipse.jface.wizard.Wizard

Implements a wizard with wizard pages from package `org.dbe.studio.tools.uigenerator.plugin.wizards.pages`.`org.dbe.studio.tools.uigenerator.plugin.wizards.pages`.

Author:

Göran Öberg

Field Summary		Page
static final boolean	**TRANSFORM_AT_NEXT**	81
static final int	**TRANSFORM_HEADER**	81

static final int	**TRANSFORM_SIZE**	*81*
static final int	**TRANSFORM_THEME**	*81*

Fields inherited from class org.eclipse.jface.wizard.Wizard
DEFAULT_IMAGE

Constructor Summary	Page
UIGeneratorWizard(org.eclipse.core.resources.IFile selectedFile) Constructs a wizard with the specified file as input.	*81*

Method Summary		Page
void	**nextPressed**(org.eclipse.jface.wizard.IWizardPage wizardPage) Perform any actions between pages, currently none.	*81*
boolean	**performFinish**() Does the final transform using the source file and each ParamaterData object.	*81*
void	**viewRefresh**()	*81*

Methods inherited from class org.eclipse.jface.wizard.Wizard
addPage, addPages, canFinish, createPageControls, dispose, getContainer, getDefaultPageImage, getDialogSettings, getNextPage, getPage, getPageCount, getPages, getPreviousPage, getShell, getStartingPage, getTitleBarColor, getWindowTitle, isHelpAvailable, needsPreviousAndNextButtons, needsProgressMonitor, performCancel, performFinish, setContainer, setDefaultPageImageDescriptor, setDialogSettings, setForcePreviousAndNextButtons, setHelpAvailable, setNeedsProgressMonitor, setTitleBarColor, setWindowTitle

Methods inherited from interface org.eclipse.jface.wizard.IWizard
addPages, canFinish, createPageControls, dispose, getContainer, getDefaultPageImage, getDialogSettings, getNextPage, getPage, getPageCount, getPages, getPreviousPage, getStartingPage, getTitleBarColor, getWindowTitle, isHelpAvailable, needsPreviousAndNextButtons, needsProgressMonitor, performCancel, performFinish, setContainer

Field Detail

TRANSFORM_AT_NEXT

```
public static final boolean TRANSFORM_AT_NEXT
```

TRANSFORM_HEADER

```
public static final int TRANSFORM_HEADER
```

TRANSFORM_THEME

```
public static final int TRANSFORM_THEME
```

TRANSFORM_SIZE

```
public static final int TRANSFORM_SIZE
```

Constructor Detail

UIGeneratorWizard

```
public UIGeneratorWizard(org.eclipse.core.resources.IFile selectedFile)
```

Constructs a wizard with the specified file as input.

Method Detail

performFinish

```
public boolean performFinish()
```

Does the final transform using the source file and each ParamaterData object.

Specified by:

`performFinish` in interface `org.eclipse.jface.wizard.IWizard`

Overrides:

`performFinish` in class `org.eclipse.jface.wizard.Wizard`

Returns:

true if transform succeded

See Also:

`org.eclipse.jface.wizard.IWizard.performFinish()`

---

nextPressed

```
public void nextPressed(org.eclipse.jface.wizard.IWizardPage wizardPage)
```

Performn any actions between pages, currently none.

Parameters:

`wizardPage` - wizard page doing the switch between pages

---

viewRefresh

```
public void viewRefresh()
```

Class UIGeneratorWizardDialog

org.dbe.studio.tools.uigenerator.plugin.wizards

```
java.lang.Object
  └org.eclipse.jface.window.Window
      └org.eclipse.jface.dialogs.Dialog
          └org.eclipse.jface.dialogs.TrayDialog
              └org.eclipse.jface.dialogs.TitleAreaDialog
                  └org.eclipse.jface.wizard.WizardDialog
                      └org.dbe.studio.tools.uigenerator.plugin.wizards.UIGeneratorWizardDialog
```

All Implemented Interfaces:

org.eclipse.jface.dialogs.IPageChangeProvider, org.eclipse.jface.operation.IRunnableContext, org.eclipse.jface.window.IShellProvider, org.eclipse.jface.wizard.IWizardContainer, org.eclipse.jface.wizard.IWizardContainer2

---

public class **UIGeneratorWizardDialog**

extends org.eclipse.jface.wizard.WizardDialog

Constructs a wizard dialog.

Author:

Göran Öberg

Inner classes inherited from class org.eclipse.jface.window.Window
`Window.IExceptionHandler`

Fields inherited from class org.eclipse.jface.wizard.WizardDialog
`WIZ_IMG_ERROR`

Fields inherited from class org.eclipse.jface.dialogs.TitleAreaDialog
`DLG_IMG_TITLE_BANNER, DLG_IMG_TITLE_ERROR, INFO_MESSAGE, WARNING_MESSAGE`

Fields inherited from class org.eclipse.jface.dialogs.Dialog
`blockedHandler, buttonBar, DIALOG_DEFAULT_BOUNDS, DIALOG_PERSISTLOCATION, DIALOG_PERSISTSIZE, DLG_IMG_ERROR, DLG_IMG_HELP, DLG_IMG_INFO, DLG_IMG_MESSAGE_ERROR, DLG_IMG_MESSAGE_INFO, DLG_IMG_MESSAGE_WARNING, DLG_IMG_QUESTION, DLG_IMG_WARNING, ELLIPSIS`

Fields inherited from class org.eclipse.jface.window.Window
`CANCEL, OK`

Constructor Summary	Page
UIGeneratorWizardDialog(org.eclipse.swt.widgets.Shell parentShell, org.eclipse.jface.wizard.IWizard newWizard)	81

Methods inherited from class org.eclipse.jface.wizard.WizardDialog
addPageChangedListener, close, getCurrentPage, getSelectedPage, removePageChangedListener, run, setMinimumPageSize, setMinimumPageSize, setPageSize, setPageSize, showPage, updateButtons, updateMessage, updateSize, updateTitleBar, updateWindowTitle

Methods inherited from class org.eclipse.jface.dialogs.TitleAreaDialog
setErrorMessage, setMessage, setMessage, setTitle, setTitleAreaColor, setTitleImage

Methods inherited from class org.eclipse.jface.dialogs.TrayDialog
close, closeTray, getTray, isDialogHelpAvailable, isHelpAvailable, openTray, setDialogHelpAvailable, setHelpAvailable

Methods inherited from class org.eclipse.jface.dialogs.Dialog
applyDialogFont, close, convertHeightInCharsToPixels, convertHorizontalDLUsToPixels, convertVerticalDLUsToPixels, convertWidthInCharsToPixels, create, getBlockedHandler, getImage, setBlockedHandler, shortenText

Methods inherited from class org.eclipse.jface.window.Window
close, create, getDefaultImage, getDefaultImages, getDefaultOrientation, getReturnCode, getShell, getWindowManager, open, setBlockOnOpen, setDefaultImage, setDefaultImages, setDefaultModalParent, setDefaultOrientation, setExceptionHandler, setWindowManager

Methods inherited from interface org.eclipse.jface.window.IShellProvider
getShell

Methods inherited from interface org.eclipse.jface.wizard.IWizardContainer2
updateSize

Methods inherited from interface org.eclipse.jface.wizard.IWizardContainer
getCurrentPage, getShell, showPage, updateButtons, updateMessage, updateTitleBar,

updateWindowTitle

Methods inherited from interface org.eclipse.jface.operation.IRunnableContext
run

Methods inherited from interface org.eclipse.jface.dialogs.IPageChangeProvider
addPageChangedListener, getSelectedPage, removePageChangedListener

Constructor Detail

UIGeneratorWizardDialog

```
public UIGeneratorWizardDialog(org.eclipse.swt.widgets.Shell parentShell,
                               org.eclipse.jface.wizard.IWizard newWizard)
```

Package org.dbe.studio.tools.uigenerator.plugin.wizards.pages

Provides two classes for each wizard page, one constructing the wizard GUI and one encapsulating its data.

See:

Description

Interface Summary		Page
IParameterData	Interface for classes offering XML-serializable data objects.	81

Class Summary		Page
EnvData	Implements a serializable store for the rpc environment address in domain/class name format.	81
FormatData	Implements a serializable store for the OpenLaszlo theme name and user interface size to use for the generated GUI.	81
FormatWizardPage	Implements a wizard page for selecting OpenLaszlo user interface theme and size.	81
HeaderData	Implements a serializable store for GUI header components.	81
HeaderWizardPage	Builds GUI for defining header parameters in generated GUI.	81

XMLSerializable	To be extended by classes that need a simple way to serialise themselves to XML.	*81*

Package org.dbe.studio.tools.uigenerator.plugin.wizards.pages Description

Provides two classes for each wizard page, one constructing the wizard GUI and one encapsulating its data.

Class EnvData

org.dbe.studio.tools.uigenerator.plugin.wizards.pages

```
java.lang.Object
  └ org.dbe.studio.tools.uigenerator.plugin.wizards.pages.XMLSerializable
      └ org.dbe.studio.tools.uigenerator.plugin.wizards.pages.EnvData
```

All Implemented Interfaces:

Serializable

---

public class **EnvData**

extends XMLSerializable

implements Serializable

Implements a serializable store for the rpc environment address in domain/class name format.

Author:

Göran Öberg

Constructor Summary	*Page*
EnvData() Simple constructor.	*81*
EnvData(String className) Constructor given initial class name.	*81*

Method Summary		Page
String	**getClassName**() Get the class name.	81
void	**setClassName**(String className) Set the class name.	81

Methods inherited from class org.dbe.studio.tools.uigenerator.plugin.wizards.pages.XMLSerializable
toXML

Constructor Detail

EnvData

```
public EnvData()
```

Simple constructor.

EnvData

```
public EnvData(String className)
```

Constructor given initial class name.

Method Detail

getClassName

```
public String getClassName()
```

Get the class name.

Returns:

Returns the className.

---

setClassName

```
public void setClassName(String className)
```

Set the class name.

Parameters:

`className` - The className to set.

Class FormatData

org.dbe.studio.tools.uigenerator.plugin.wizards.pages

```
java.lang.Object
  └ org.dbe.studio.tools.uigenerator.plugin.wizards.pages.XMLSerializable
      └ org.dbe.studio.tools.uigenerator.plugin.wizards.pages.FormatData
```

All Implemented Interfaces:

Serializable

---

public class **FormatData**

extends XMLSerializable

implements Serializable

Implements a serializable store for the OpenLaszlo theme name and user interface size to use for the generated GUI.

Author:

Göran Öberg

Constructor Summary	Page
FormatData() Simple constructor.	81
FormatData(String colorTheme, Integer width, Integer height) Constructor given initial theme name and size.	81

Method Summary		Page
String	**getColorTheme**()	81
Integer	**getHeight**()	81
Integer	**getWidth**()	81
void	**setColorTheme**(String colorTheme)	81
void	**setHeight**(Integer height)	81
void	**setWidth**(Integer width)	81

Methods inherited from class org.dbe.studio.tools.uigenerator.plugin.wizards.pages.XMLSerializable
toXML

Constructor Detail

FormatData

```
public FormatData()
```

Simple constructor.

FormatData

```
public FormatData(String colorTheme,
                  Integer width,
                  Integer height)
```

Constructor given initial theme name and size.

Method Detail

getColorTheme

```
public String getColorTheme()
```

Returns:

the colorTheme

---

setColorTheme

```
public void setColorTheme(String colorTheme)
```

Parameters:

`colorTheme` - the colorTheme to set

---

getHeight

```
public Integer getHeight()
```

Returns:

the height

---

setHeight

```
public void setHeight(Integer height)
```

Parameters:

`height` - the height to set

---

getWidth

```
public Integer getWidth()
```

Returns:

the width

---

setWidth

```
public void setWidth(Integer width)
```

Parameters:

`width` - the width to set

Class FormatWizardPage

org.dbe.studio.tools.uigenerator.plugin.wizards.pages

```
java.lang.Object
  └org.eclipse.jface.dialogs.DialogPage
      └org.eclipse.jface.wizard.WizardPage
          └org.dbe.studio.tools.uigenerator.plugin.wizards.pages.FormatWizardPage
```

All Implemented Interfaces:

org.eclipse.jface.dialogs.IDialogPage, org.eclipse.jface.dialogs.IMessageProvider, IParameterData, org.eclipse.jface.wizard.IWizardPage

---

public class **FormatWizardPage**

extends org.eclipse.jface.wizard.WizardPage

implements IParameterData

Implements a wizard page for selecting OpenLaszlo user interface theme and size.

Author:

Göran Öberg

Fields inherited from interface org.eclipse.jface.dialogs.IMessageProvider
ERROR, INFORMATION, NONE, WARNING

Constructor Summary	Page
FormatWizardPage(String pageName) Defines a FormatWizardPage object with the specified page name.	81
FormatWizardPage(String pageName, String title, org.eclipse.jface.resource.ImageDescriptor titleImage) Defines a FormatWizardPage object with the specified page name, title and title image.	81

Method Summary		Page
void	**createControl**(org.eclipse.swt.widgets.Composite parent) Creates the page content and controls.	81
void	**createPageContent**(org.eclipse.swt.widgets.Composite parent) Creates and places the GUI objects.	81
XMLSerializable	**getParameterData**() Returns a XML serialisable object with the current configuration data.	81

Methods inherited from class org.eclipse.jface.wizard.WizardPage
canFlipToNextPage, getImage, getName, getNextPage, getPreviousPage, getShell, getWizard, isPageComplete, setDescription, setErrorMessage, setImageDescriptor, setMessage, setPageComplete, setPreviousPage, setTitle, setWizard, toString

Methods inherited from class org.eclipse.jface.dialogs.DialogPage
dispose, getControl, getDescription, getErrorMessage, getImage, getMessage, getMessageType, getShell, getTitle, performHelp, setDescription, setErrorMessage, setImageDescriptor, setMessage, setMessage, setTitle, setVisible

Methods inherited from interface org.eclipse.jface.dialogs.IDialogPage
createControl, dispose, getControl, getDescription, getErrorMessage, getImage, getMessage, getTitle, performHelp, setDescription, setImageDescriptor, setTitle, setVisible

Methods inherited from interface org.eclipse.jface.dialogs.IMessageProvider
getMessage, getMessageType

Methods inherited from interface org.eclipse.jface.wizard.IWizardPage
canFlipToNextPage, getName, getNextPage, getPreviousPage, getWizard, isPageComplete, setPreviousPage, setWizard

Methods inherited from interface org.dbe.studio.tools.uigenerator.plugin.wizards.pages.IParameterData
getParameterData

Constructor Detail

FormatWizardPage

```
public FormatWizardPage(String pageName)
```

Defines a FormatWizardPage object with the specified page name.

FormatWizardPage

```
public FormatWizardPage(String pageName,
                        String title,
                        org.eclipse.jface.resource.ImageDescriptor titleImage)
```

Defines a FormatWizardPage object with the specified page name, title and title image.

Method Detail

createControl

```
public void createControl(org.eclipse.swt.widgets.Composite parent)
```

Creates the page content and controls.

Specified by:

`createControl` in interface `org.eclipse.jface.dialogs.IDialogPage`

Parameters:

`parent` - wizard page to create content in

---

createPageContent

```
public void createPageContent(org.eclipse.swt.widgets.Composite parent)
```

Creates and places the GUI objects.

Parameters:

`parent` - wizard page to put content into

---

getParameterData

```
public XMLSerializable getParameterData()
```

Returns a XML serialisable object with the current configuration data.

Specified by:

`getParameterData` in interface `IParameterData`

Returns:

Object containing wizard page specific data

Class HeaderData

org.dbe.studio.tools.uigenerator.plugin.wizards.pages

```
java.lang.Object
  └ org.dbe.studio.tools.uigenerator.plugin.wizards.pages.XMLSerializable
      └ org.dbe.studio.tools.uigenerator.plugin.wizards.pages.HeaderData
```

All Implemented Interfaces:

Serializable

---

public class **HeaderData**

extends XMLSerializable

implements Serializable

Implements a serializable store for GUI header components.

Author:

Göran Öberg

Constructor Summary	Page
HeaderData() Simple constructor.	81
HeaderData(String title, String body, String pic, String smid) Constructor given initial data	81

Method Summary		Page
String	**getBody**()	81
String	**getPic**()	81
String	**getSmid**()	81
String	**getTitle**()	81
void	**setBody**(String body)	81
void	**setPic**(String pic)	81
void	**setSmid**(String smid)	81
void	**setTitle**(String title)	81

Methods inherited from class org.dbe.studio.tools.uigenerator.plugin.wizards.pages.XMLSerializable
toXML

Constructor Detail

HeaderData

```
public HeaderData()
```

Simple constructor.

HeaderData

```
public HeaderData(String title,
                  String body,
                  String pic,
                  String smid)
```

Constructor given initial data

Method Detail

getBody

```
public String getBody()
```

Returns:

Returns the description text.

---

setBody

```
public void setBody(String body)
```

Parameters:

`body` - The desciption text to set.

---

getPic

```
public String getPic()
```

Returns:

Returns the image name

---

setPic

```
public void setPic(String pic)
```

Parameters:

`pic` - the image name to set

---

getTitle

```
public String getTitle()
```

Returns:

Returns the title text.

---

setTitle

```
public void setTitle(String title)
```

Parameters:

`title` - The title text to set.

---

getSmid

```
public String getSmid()
```

Returns:

the SMID

---

setSmid

```
public void setSmid(String smid)
```

Parameters:

`smid` - the SMID to set

Class HeaderWizardPage

org.dbe.studio.tools.uigenerator.plugin.wizards.pages

```
java.lang.Object
  └org.eclipse.jface.dialogs.DialogPage
      └org.eclipse.jface.wizard.WizardPage
          └org.dbe.studio.tools.uigenerator.plugin.wizards.pages.HeaderWizardPage
```

All Implemented Interfaces:

org.eclipse.jface.dialogs.IDialogPage, org.eclipse.jface.dialogs.IMessageProvider, IParameterData, org.eclipse.jface.wizard.IWizardPage

---

public class **HeaderWizardPage**

extends org.eclipse.jface.wizard.WizardPage

implements IParameterData

Builds wizard page for defining header parameters in generated GUI.

Author:

Göran Öberg

Fields inherited from interface org.eclipse.jface.dialogs.IMessageProvider
ERROR, INFORMATION, NONE, WARNING

Constructor Summary	Page
HeaderWizardPage(String pageName) Constructor with name for header of wizard page.	81
HeaderWizardPage(String pageName, String title, org.eclipse.jface.resource.ImageDescriptor titleImage) Constructor with name for header, title and image of wizard page.	81

Method Summary		Page
void	**createControl**(org.eclipse.swt.widgets.Composite parent) Creates the page content and controls.	81
void	**createPageContent**(org.eclipse.swt.widgets.Composite parent) Create widgets and put them into wizard page.	81
XMLSerializable	**getParameterData**() Creates a serializable instance with the current data.	81
String	**getSelectedPicturePath**() Get path to picture file selected in wizard.	81

Methods inherited from class org.eclipse.jface.wizard.WizardPage
canFlipToNextPage, getImage, getName, getNextPage, getPreviousPage, getShell, getWizard, isPageComplete, setDescription, setErrorMessage, setImageDescriptor, setMessage, setPageComplete, setPreviousPage, setTitle, setWizard, toString

Methods inherited from class org.eclipse.jface.dialogs.DialogPage
dispose, getControl, getDescription, getErrorMessage, getImage, getMessage, getMessageType, getShell, getTitle, performHelp, setDescription, setErrorMessage, setImageDescriptor, setMessage, setMessage, setTitle, setVisible

Methods inherited from interface org.eclipse.jface.dialogs.IDialogPage
createControl, dispose, getControl, getDescription, getErrorMessage, getImage, getMessage, getTitle, performHelp, setDescription, setImageDescriptor, setTitle, setVisible

Methods inherited from interface org.eclipse.jface.dialogs.IMessageProvider
getMessage, getMessageType

Methods inherited from interface org.eclipse.jface.wizard.IWizardPage
canFlipToNextPage, getName, getNextPage, getPreviousPage, getWizard, isPageComplete, setPreviousPage, setWizard

Methods inherited from interface org.dbe.studio.tools.uigenerator.plugin.wizards.pages.IParameterData
getParameterData

Constructor Detail

HeaderWizardPage

```
public HeaderWizardPage(String pageName)
```

Constructor with name for header of wizard page.

HeaderWizardPage

```
public HeaderWizardPage(String pageName,
                        String title,
                        org.eclipse.jface.resource.ImageDescriptor titleImage)
```

Constructor with name for header, title and image of wizard page.

Method Detail

createControl

```
public void createControl(org.eclipse.swt.widgets.Composite parent)
```

Creates the page content and controls.

Specified by:

`createControl` in interface `org.eclipse.jface.dialogs.IDialogPage`

Parameters:

`parent` - wizard page to create content in

---

createPageContent

```
public void createPageContent(org.eclipse.swt.widgets.Composite parent)
```

Create widgets and put them into wizard page.

Parameters:

`parent` - wizard page to put content into

getSelectedPicturePath

```
public String getSelectedPicturePath()
```

Get path to picture file selected in wizard.

Returns:

path to selected picture.

---

getParameterData

```
public XMLSerializable getParameterData()
```

Creates a serializable instance with the current data.

Specified by:

`getParameterData` in interface `IParameterData`

Returns:

Object containing wizard page specific data

See Also:

`EnvData`, `XMLSerializable`

Interface IParameterData

org.dbe.studio.tools.uigenerator.plugin.wizards.pages

All Known Implementing Classes:

FormatWizardPage, HeaderWizardPage

---

public interface **IParameterData**

Interface for classes offering XML-serializable data objects.

Author:

Göran Öberg

Method Summary		Page
XMLSerializable	**getParameterData**() Returns an Object with the wizard page's data in an XMLSerializable object.	81

Method Detail

getParameterData

```
public XMLSerializable getParameterData()
```

Returns an Object with the wizard page's data in an XMLSerializable object.

Returns:

Object containing wizard page specific data

Class XMLSerializable

org.dbe.studio.tools.uigenerator.plugin.wizards.pages

```
java.lang.Object
  └ org.dbe.studio.tools.uigenerator.plugin.wizards.pages.XMLSerializable
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

EnvData, FormatData, HeaderData

---

public class **XMLSerializable**

extends Object

implements Serializable

To be extended by classes that need a simple way to serialise themselves to XML.

Author:

Göran Öberg

Constructor Summary	Page
XMLSerializable()	81

Method Summary		Page
final ByteArrayOutputStream	toXML() Encodes the object in XML	81

Constructor Detail

XMLSerializable

```
public XMLSerializable()
```

Method Detail

toXML

```
public final ByteArrayOutputStream toXML()
```

Encodes the object in XML

Returns:

ByteArrayOutputStream containing object encoded in XML

Package org.dbe.studio.tools.uigenerator.popup.actions

Provides for the connection to the Eclipse content menu, invoking the DBE UI Generator Wizard.

See:

Description

Class Summary		Page
UIGeneratorPopupAction	Handles content menu connection for DBE Studio UI Generator.	81

Package org.dbe.studio.tools.uigenerator.popup.actions Description

Provides for the connection to the Eclipse content menu, invoking the DBE UI Generator Wizard.

Class UIGeneratorPopupAction

org.dbe.studio.tools.uigenerator.popup.actions

```
java.lang.Object
  └ org.dbe.studio.tools.uigenerator.popup.actions.UIGeneratorPopupAction
```

All Implemented Interfaces:

org.eclipse.ui.IActionDelegate, org.eclipse.ui.IObjectActionDelegate

---

public class **UIGeneratorPopupAction**

extends Object

implements org.eclipse.ui.IObjectActionDelegate

Handles content menu connection for DBE Studio UI Generator.

Author:

Göran Öberg

Constructor Summary	Page
UIGeneratorPopupAction() Simple constructor.	81

Method Summary		Page
static org.eclipse.swt.widgets.Shell	getShell() Acquires a shell to put the wizard in.	81

void	**run**(org.eclipse.jface.action.IAction action) Reacts to menu selection on content menu and initiates wizard based on provided action.	81
void	**selectionChanged**(org.eclipse.jface.action.IAction action, org.eclipse.jface.viewers.ISelection selection) Handle and store selectionChanged event.	81
void	**setActivePart**(org.eclipse.jface.action.IAction action, org.eclipse.ui.IWorkbenchPart targetPart)	81

Methods inherited from interface org.eclipse.ui.IObjectActionDelegate
setActivePart

Methods inherited from interface org.eclipse.ui.IActionDelegate
run, selectionChanged

Constructor Detail

UIGeneratorPopupAction

```
public UIGeneratorPopupAction()
```

Simple constructor.

Method Detail

setActivePart

```
public void setActivePart(org.eclipse.jface.action.IAction action,
                          org.eclipse.ui.IWorkbenchPart targetPart)
```

Specified by:

`setActivePart` in interface `org.eclipse.ui.IObjectActionDelegate`

See Also:

`org.eclipse.ui.IObjectActionDelegate.setActivePart(IAction, IWorkbenchPart)`

run

```
public void run(org.eclipse.jface.action.IAction action)
```

Reacts to menu selection on content menu and initiates wizard based on provided action.

Specified by:

`run` in interface `org.eclipse.ui.IActionDelegate`

See Also:

`org.eclipse.ui.IActionDelegate.run(IAction)`

---

getShell

```
public static org.eclipse.swt.widgets.Shell getShell()
```

Acquires a shell to put the wizard in.

Returns:

shell to place wizard in

---

selectionChanged

```
public void selectionChanged(org.eclipse.jface.action.IAction action,
                             org.eclipse.jface.viewers.ISelection selection)
```

Handle and store selectionChanged event.

Specified by:

`selectionChanged` in interface `org.eclipse.ui.IActionDelegate`

Parameters:

`action` - event

`selection` - current selection

See Also:

`org.eclipse.ui.IActionDelegate.selectionChanged(IAction, ISelection)`