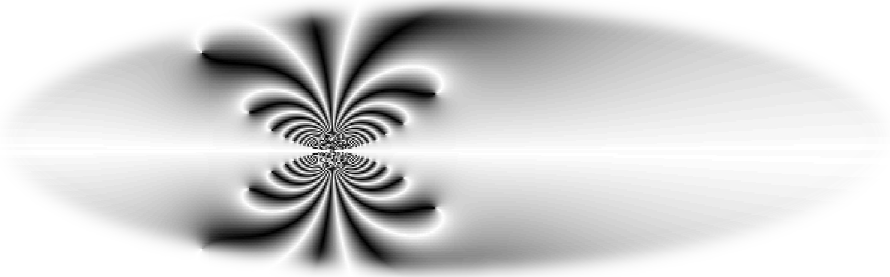
INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa



A Pattern-Language for Developing Web Applications

Luís Filipe Susano de Oliveira João

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Orientador:	Prof. Miguel Leitão Bignolas Mira da Silva
Vogal:	Prof. Ademar Manuel Teixeira de Aguiar

Setembro 2008

Agradecimentos

Gostaria de começar por agradecer ao Prof. Miguel Mira da Silva pela oportunidade de elaborar esta tese na OutSystems, pelas sugestões valiosas da sua organização e estrutura, e pela disponibilidade demonstrada durante todo este trabalho.

Agradeço ao Rodrigo Castelo, pelo acompanhamento constante, pelo apoio incondicional e por considerar um "co-autor" deste trabalho. Ao Lúcio Ferrão e à Irene Montenegro pelo entusiasmo e conselhos preciosos que muito valorizaram este projecto. Á "Green Team" (Miguel Melo, Ricardo Ferreira, Luís Pista e João Proença) pela disponibilidade e boa disposição epidémica. Estendo ainda estes agradecimentos aos meus colegas Rui Francisco, João Jesus e João Rosado, que, tal como eu, viveram esta experiência inesquecível na OutSystems, participando sempre com sugestões e questões oportunas.

Ao meu amigo e sócio da nossa recém-formada empresa Byclosure, Vasco Andrade e Silva, pelas reflexões constantes e valiosas que sempre procurou transmitir em qualquer altura deste trabalho.

Aos meus amigos Rui Pascoal, Renato Sousa, Carlos Calisto e Joana Sismeiro, que comigo partilharam estes últimos anos da minha vida estudantil e a quem devo uma profunda palavra de gratidão.

Aos participantes da VikingPLoP 2008 por lerem o meu trabalho, pelo feedback fabuloso e pelos óptimos momentos onde reflectimos sobre padrões. Ao James Coplien pela sabedoria que me procurou transmitir sobre padrões e, especialmente, linguagens de padrões. O seu conhecimento sobre esta matéria é de um valor incalculável.

À minha família pelo apoio que, por causa desta tese, foi mais remoto que presencial. E a ti, pelo brilho que emanas e pelo sorriso que contagias a cada dia que passa.

Abstract

The purpose of this thesis is to provide a mental framework of patterns to define friendly Web2.0 storage and retrieval applications. Patterns allow us not just to improve our development processes but are also a good way of documenting knowledge. A study was endorsed to prove that implementing interaction design patterns are the key for productivity enhancement.

The development of such applications should follow a pattern-driven paradigm where the applications are implemented by composing patterns. To achieve such a goal, a pattern-language was defined together with a set of patterns previously identified in enterprise applications, rich in information storage and retrieval. This language syntax is displayed by a pattern format that specifies built-to-change patterns and supports their composition.

As a prove of concept, we went through a case study for testing the pattern-language. A simple application was developed followed a pattern-driven web development based on interaction design patterns.

Keywords: *Interaction Design Patterns, Pattern-language, Storage and Retrieval Patterns*

Resumo

O objectivo desta tese é fornecer uma *framework* mental de padrões que possibilitem definir uma aplicação Web 2.0 centrada no armazenamento e recolha de dados. Os padrões permitem-nos não só melhorar os nossos processos de desenvolvimento como também são uma boa solução para documentar conhecimento. Neste trabalho provou-se que implementando padrões de desenho de interfaces se atingirá um maior aumento de produtividade.

O desenvolvimento deste tipo de aplicações deve seguir um paradigma orientado aos padrões onde as aplicações são desenvolvidas através da composição destes. Para atingir esse objectivo, uma linguagem de padrões foi definida através de um conjunto de padrões, previamente identificado em aplicações empresariais ricas em armazenamento e recolha de dados. A sintaxe desta linguagem é definida através de um formato especial para definir padrões que suporta as composições e gere facilmente a mudança dos mesmos.

Como demonstração de valor e avaliação deste trabalho, testou-se a linguagem de padrões através de um caso de estudo. Uma simples aplicação foi desenvolvida, seguindo o paradigma orientado a padrões de desenvolvimento Web, tendo como padrões base os de design de interacções.

Palavras-chave: *Padrões de Interacção, Linguagem de Padrões, Padrões de Aplicações Empresariais*

Table of Contents

List of Figures

List of Tables

Acronyms and Abbreviations

CMS	Content Management System
CRM	Customer Relationship Management
CRUD	Create, Read, Update, Delete
EAA	Enterprise Application Architecture
GoF	Gang of Four
HCI	Human Computer Interaction
OML	OutSystems Markup Language
PLML	Pattern Language Mark-up Language
PoEAA	Patterns of Enterprise Application Architecture
POSA	Pattern-Oriented Software Architecture
UED	User Experience & Design
UI	User Interface
UX	User Experience

1 Introduction

This thesis is about creating a pattern language to define friendly Web2.0 storage and retrieval applications.

Everybody knows the world is made up of processes from which patterns emerge. Many fields use patterns in various ways: in music and literature, a pattern is the coherent structure or design of a song or book. In art, a pattern is the composition or plan of a work of graphic or plastic art. In architecture, a pattern is an architectural design or style. In chess, a pattern is a set of moves that may be applied in an overall strategy [1].

1.1 Context

This thesis was written in the OutSystems Company. OutSystems sells a product for developing web applications based on an end-to-end visual development environment. A developer can create and compose applications using a highly intuitive and visual environment. User interaction flows, data models, business rules, scheduled processes, web services, and integration adapters can be used to create applications using an intuitive drag-and-drop process.

OutSystems currently uses a style guide which contains a series of screens that help developing applications faster while retaining the OutSystems look and feel. It aims to avoid basic design mistakes in terms of Font and Spacing decisions, promote common feel and pretty look among all applications created in OutSystems, simplify the customization of the look and feel of template solutions to match the requirements of each specific customer. It is a way to document knowledge and provide best practices among the developers along the organization.

However, style guide does not contain solutions for most of the problems. At the same time, developing based on style guide is about “copy pasting” instead of giving semantic and expressiveness to new elements in the OutSystems language.

1.2 Problem

Staying competitive is about reducing development cost. In that sense, a first approach was to develop a platform for modeling and execution business processes. However, we soon realize that this tool didn’t enhance the maximum productivity as the implementation of patterns in the OutSystems platform. We prove this statement in chapter 3.

Another problem inside the organization is how to increase expressiveness of OutSystems' language, in order to increase productivity and reduce costs. Reducing the development effort, development skills, maintenance effort and operation effort are the key problems of this thesis that are solved through patterns.

1.3 Proposal

Our goal is to reduce the development by documenting several patterns in a formal way. The implementation will not be addressed. Patterns are solutions for recurring problems.

Firstly, it is important to discover which patterns provide higher returns (in terms of productivity enhancement) and which ones we should catalog. Furthermore, we must choose way to document the patterns and define a shared vocabulary for them.

Patterns and pattern languages offer an approach to design with much potential. Research in these areas is now needed to ensure that this promise is fulfilled and that pattern language research makes an effective and lasting contribution to the practice and understanding of interaction design.

1.4 Organization/Structure

This report is composed by 6 chapters:

1. Introduction – We describe the context, problem and goals of this study;
2. State of the Art – We reference a set of popular papers and make a summary of research in patterns, patterns languages and interaction design patterns;
3. Stereotype Patterns – Description of high-level application patterns for defining our target and justify our path;
4. Pattern-Language for Storage and Retrieval Applications – We describe our mining process for the construction of a pattern language, we define a format to describe our patterns and then we specify some examples of patterns discovered and documented;
5. Testing the framework - Visual development of a simple web application using pattern-driven development (with patterns described in the previous chapter);
6. Conclusions – We present some thoughts about this work and future research.

2 State of the Art

The goal of this section is to provide an overview over software patterns. Firstly, we introduce the notions of pattern and pattern language. We describe the various types of patterns and tips for identifying them. Then, we go through some well-known formats for representing patterns. We close the chapter by explaining how patterns can be organized and connected.

2.1 What is a pattern?

In the mid-70's, patterns were introduced by Christopher Alexander in his books about architectural building [2] [3]. Alexander noticed that certain solutions always apply to the same recurring problems and developed patterns as a design knowledge documentation method. He defines a pattern as "…*a problem which occurs over and over again in our 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 the same way twice*" [3].

Software Engineering adopted patterns – with "Gang of Four" (GoF) [4] and others [5] [6] – as a way to describe recurring problems and their best known solutions, and facilitate reuse of software. Software patterns were adopted to allow sharing of larger units, and they specify in quite fine detail how components interact. As such they are much more prescriptive than patterns for architecture.

User interface designers also noticed that certain design problems occurred over and over [7]. These problems generally have known good solutions. However, there has been a problem communicating them and there is a huge effort on studying effective use and reuse of Human Computer Interaction (HCI) Knowledge [8]. Guidelines represent a possible solution, but they are generally seen as hard to interpret and requiring excessive effort to find relevant material [9]. For this reason, there has been an increasing interest in patterns to document user-interface design solutions.

2.1.1 **Pattern Languages and Pattern Catalogues**

Alexander's original work was not merely about individual patterns, but was explicitly concerned with the concept of pattern languages. Taken in isolation, patterns are, at best, "*unrelated good ideas*" [3].

The idea behind a pattern language is that a body of patterns is presented with a structure that guides from pattern to pattern. It begins with (usually) some very strategic patterns, and then each pattern leads to a point where we have to decide to apply other patterns. A pattern language has a flow that connects the various patterns [10].

A key concept in distinguishing pattern collections from pattern languages is the idea of generativity. One reading of the organization of *A Pattern Language* [3] suggests the idea of generating designs by implicit sequencing of decisions, derived by traversing the network of links between the individual patterns [7].

A pattern language helps designers to ask and answer the right question at the right time, i.e. the language can be used to sequence design decisions [7].

Pattern languages are very hard to write [10]. None of the known literature describes patterns as a language, but instead as a catalog of patterns [4] [5] [6]. However, some research has been done on language patterns specification, especially in Interaction Design [11].

A significant outcome of the CHI2003 workshop is the Pattern Language Markup Language (PLML) specification [12]. The goal in deriving PLML was to bring order to the many (inconsistent) forms pattern researchers had proposed and used. The purpose was seeking a way in which patterns and pattern languages from various authors could refer to patterns in other collections – perhaps even combined into larger meta-collections [12].

2.1.2 Characteristics and Usage of Patterns

Patterns provide a way to organize and name those ordinary solutions to make it easier for people to use them. Since these solutions are ordinary, it's common that experts in a field won't find anything new in a patterns book. For such people the biggest value of a patterns book is to help them to pass on the solutions to their colleagues [10].

The whole point of writing a pattern is to describe a recurring and useful solution. Success is all about doing that in a way that others can replicate that solution when it's appropriate. Everything else is secondary - which means that however we write the pattern, whatever form we take - all has to support this [10].

Understanding the problem (or problems, as patterns can solve more than one) is a key part of understanding the solution. Thinking about the problem helps focusing on the “core of the solution”. It also helps keeping from sliding too far into a tools-oriented discussion. So understanding the problem is important - indeed vital. But the solution should remain the focus of the pattern [10].

For the GoF [4], the rationales behind design patterns were to:

- Provide designers with a shared vocabulary to discuss and comment on design alternatives (*lingua franca*);

- Provide designers with micro-architecture building blocks that they can compose to generate more complex architectures;
- Make it easy to learn new frameworks by referring to design patterns in the framework's description;
- Discuss the trade-offs that are related to a specific design decision [4].

For Fincher [13], patterns are notable because they are based on examples, facilitate multiple levels of abstraction, bridge the gap between the physical and the social aspects of design, and are amenable to piecemeal development. He also identifies capture of practice and abstraction as important, but adds: organizing principle to relate patterns to other patterns in a way that enables design; a value system that is embodied in the patterns; and a particular presentational style [7].

Fowler also suggests using his patterns in collaboration with requirements analysts, clients and domain experts to develop specific models for particular projects [6].

A variation on the use of patterns concerning paper prototyping is work by Lin and Landay [14] who propose to integrate patterns into a design sketching environment, allowing designers to drag and drop patterns into their sketches and customize them to meet local requirements. While this approach is intended for experienced designers, its potential application within participatory design to support early prototyping with patterns is clear [7].

May and Taylor [15] propose patterns as a tool for organizational knowledge management. In Human Computer Interaction (HCI), Henniger [16] suggests a process where each development project begins by interrogating a corporate memory to retrieve and select patterns (and guidelines) to use within the project [7].

Some authors have investigated incorporating software patterns into development tools, or implementing patterns as components of programming languages. This has also been proposed in interaction design [14] [17]. It can be objected that such efforts only incorporate the 'solution' part of the pattern, but do not provide advice to software designers about when to use that particular pattern [7].

2.1.3 **Example of a Pattern**

Filter Pattern - A filter is a condition for searching objects. In information systems, users need very frequently specific searching tools. In the conceptual phase, analysts must capture such requirement. [17]

Name	Filter
Also known as	Query
Problem	The user needs to browse and search objects belonging to a large set.
Context	In information systems is a very frequent task to search for objects. Powerful search mechanisms are needed to help the user.
Forces	The number of objects in the set may hinder the searching process. A complex query interface can be hard to understand for not experienced users.
Solution	Provides a mechanism to query the objects satisfying certain conditions. The analyst can express it in a OQL-like syntax with variables, letting the user introduce data in such variables in run time.
Restrictions	Objects to be searched must comparable (in other words, objects have a common type to be comparable).
Example	Web searching engines, library searching facilities, etc.
Rationale	Provides a mechanism to reduce complexity. The user can incrementally narrow the searching scope.
Related Patterns	Order Criterium, Display Set, Population Observation.

Table 1– Specification of a Filter Pattern [17]

2.2 Patterns Taxonomy

The primary focus of this thesis is on patterns and pattern languages that discuss interaction and interface design issues. There are, however, a large number of patterns from other domains, e.g. software engineering and organizational design, which may have a bearing on interactions between humans and computers. To avoid extending the scope of our review beyond practical limits, Dearden and Finlay [7] define three broad classes of software-related pattern and pattern language: general design patterns, interfaces, patterns in the application domain.

This taxonomy doesn't have granularity into consideration. It just structures software patterns according to their behavior and purpose.

2.2.1 General Design Patterns

A problem is stated in terms of desirable qualities of the internal structure and behavior of software, and the solution is stated in terms of suggested code structures [7].

Martin Fowler [18] divides general design patterns in: Enterprise Application Architecture Patterns, Enterprise Integration Patterns and Domain Logic Patterns.

Enterprise Application Architecture (EAA) Patterns

Enterprise Application is the name Fowler gives to a certain class of software systems: the data intensive software systems on which so many businesses run.

Most books on EAA begin by breaking an enterprise application into logical layers. This layering structure then drives other design decisions within and between the layers. As such it's no surprise that patterns tend to be similarly organized through layers. Each author has their own layering structure, but there are recognizable similarities between the layering structures [18].

Examples of this kind of patterns are the ones found in Patterns of Enterprise Application Architecture [19], Microsoft Enterprise Solution Patterns [20], Core J2EE Patterns [21] or Design Patterns: Elements of Reusable Object-Oriented Software [4].

Note that Enterprise Architecture is quite different from EAA. EAA deals with the design of enterprise applications. Enterprise Architecture deals with the organization of multiple applications in an enterprise into a coherent whole [18].

Enterprise Integration Patterns

Enterprise Applications are somewhat independent applications, but to function they need to work together. Stitching together independently developed Enterprise Applications is the work of integration. Often there is a need to integrate applications that weren't design with any integration in mind, let alone the specific one that one is using, or they expect to integrate using a new technology [18].

Enterprise Integration Patterns [22] and Microsoft Integration Patterns [23] are examples of enterprise integration patterns.

Domain Logic Patterns

One of the most important, yet often forgot, aspects of enterprise applications is the domain logic. These are the business rules, validations, and calculations that operate on the data as it is brought into an information system or displayed by it. For simple database filing systems, there is often little or no domain logic. However, for many systems there is often quite complex domain logic, and this logic is subject to regular change as business conditions change [18].

Examples of domain logic patterns can be found in: Domain-Driven Design [24], Data-Model Patterns [25], and Patterns of Enterprise Application Architecture [19].

2.2.2 Patterns in the Application Domain

A problem is stated in the domain of desirable interaction behaviors, and the solution is stated in terms of suggested code structures.

Patterns very dependent on the application domain are those for implementing systems that follow a "tools and materials"; for implementing digital sound synthesis systems; for implementing queuing of interaction patterns for e-commerce agent systems and patterns for mobile services [7].

2.2.3 Patterns in Interaction Design

A problem is stated in the domain of human interaction issues, and the solution is stated in terms of suggested perceivable interaction behavior.

Examples of catalogs of these patterns can be found in: Yahoo Design Pattern Library [26], Welie Patterns [27], User Interface Design Patterns Library [28] [29] and Ajax Design Patterns [30].

2.3 Identifying Patterns

It is usually agreed that patterns must be discovered by reference to design solutions, rather than being constructed from first principles. Alexander suggests that "patterns are found by trial and error and by observation" [3]. Coad [31] discusses "discovering" patterns from experience and Gabriel [32] uses the metaphor of "mining" patterns from existing designs [7].

One element that is perhaps unique to interaction design patterns is the need to include the notion of temporality [33]. Unlike architecture, HCI deals with an artifact where time is significant and the context of, and solutions to, interaction problems are liable to be dynamic rather than static. A pattern must therefore be able to capture this temporal interactive element. The use of alternative media (such as video) has been suggested to illustrate interactive time-based solutions [33], but the fundamental issue of abstracting true interaction rather than simply snapshots of appearance or behavior remains [7].

On the other hand, patterns should also embody a timeless quality, presenting a solution that is applicable regardless of platform or technology. This is arguably a weakness in many current interaction design patterns, which are strongly based on a particular and current user interface paradigm (graphical user interfaces for example). It is suggested that patterns that address interaction issues at a "high level" of abstraction may be timeless, but that patterns that are closer to the detail of interaction design perhaps necessarily reflect current paradigms [7].

2.3.1 Real-World Experiences on Identifying Patterns

There have been some efforts concerning identification of patterns in popular websites.

A report [34] describes a detailed study of thee museum websites in order to determine the general characteristics and issues in museum site design. The study was also used to refine existing patterns where needed as well as to create new patterns to describe design solutions that weren't previously described in the pattern collection.

A 'drill-down' method was used to analyze the sites: starting from the homepage, all major sections were reviewed followed by an examination of specific sections such as the search engine or online shop. The end of the analysis focused on writing the *Museum Site* pattern. This high-level pattern discusses the main ingredients of a museum site and points to other relevant patterns.

A UIE Report [35] brings such core design concepts to the surface. The series presents proven, time-tested ideas that drive today's most successful designs. In Web Application Structure [35], Hagan Rivers, a pioneer web application developer, takes a closer look at the navigation and orientation elements of web applications. The main goal is to pioneer web application designers, examine seven unique web applications, and highlight the most interesting design elements.

At Yahoo! [36], a pattern most often comes into the library via the traditional design process. Within the context of a product design cycle, a solution to the common problem is created.

Design research and designers collaborate and will test the range of low-fidelity prototypes to final product usability testing. The designer of the solution, or the central Yahoo! UED (User Experience & Design) group recognizing solutions to common problems across the network, writes the pattern for submission to the library. Additionally, the central UED design research team periodically reviews research from all across Yahoo! and makes recommendations for refinements to the pattern. The pattern is then edited, published, reviewed and labeled with an adherence rating. As designs evolve and technologies change that enable new solutions to emerge, the pattern library [26] evolves as well.

2.4 Representation of Patterns

Every author tends to make his own particular pattern form, but certain pattern forms have become more well-known. These are often used exactly by new authors, or at least as starting points [10].

2.4.1 Alexandrian Form

The Alexandrian form [3] is a very narrative form, with relatively few headings. As a result it tends to flow better than most alternatives when one reads it. The bolded summary sentences of the problem and the

solution stand out well, and allow you to skip through a large body of patterns very quickly [10]. As well as the patterns in Alexander's book [3], good examples of this form can also be found in Domain-Driven Design [24].

2.4.2 GoF Form

The GoF [4] is a very structured form, breaking up the pattern into many headings: Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. The GoF patterns are quite large, a dozen pages each [10].

2.4.3 POSA Form

Similarly to GoF, POSA [5] is a very structured and quite large form, although the headings are different: summary, example, context, problem, solution, structure, dynamics, implementation, example resolved, variants, known uses, consequences, and see also. The patterns are usually just over a dozen pages in length. An important part of this form is that the patterns are preceded by a narrative chapter that summarizes the patterns and describes the overall topic [10].

2.4.4 PoEAA Form

It's fairly narrative, with a few sections: how it works, when to use it, and one or more examples. The length averages eight pages, but it varies from one page to well over a dozen [10].

2.4.5 Formal Form

Brochers [33] suggests a formal hypertext model of a pattern language. A formal description of patterns makes it less ambiguous for the parties involved to decide what a pattern is supposed to look like, in terms of structure and content. It also makes it possible to design computer based tools that help authors in writing, and readers in understanding patterns. The formal syntactic definition is described below:

- A pattern language is a directed acyclic graph (DAG) **PL = (P, R)** with nodes $\mathrm{P} = \{P_1, ..., P_n\}$ and edges $\mathrm{R} = \{R_1, \ldots, R_m\}$
- Each node $P \in \mathrm{P}$ is called a pattern
- For $P, Q \in \mathrm{P} : P \; references \; Q \Leftrightarrow \exists R = (P, Q) \in \mathrm{R}$
- The set of edges leaving a node $P \in \mathrm{P}$ is called its *references*. The set of edges entering it is called its *context*

- Each node $P \in \mathrm{P}$ is itself a set $P = \{n, r, i, p, f_1 ... f_i, e_1 ... e_j, s, d\}$ of a name *n*, ranking *r*, illustration *i*, problem *p* with forces $f_1 ... f_j$ examples $e_1 ... e_j$ the solution *s* and diagram *d*

This syntactic definition is augmented with the following semantics [33]:

- Each pattern of a language captures a recurring design problem, and suggests a proven solution to it.
- Each pattern has a context represented by edges pointing to it from higher-level patterns.
- They sketch the design situations in which it can be used. Similarly, its references show what lower-level patterns can be applied after it has been used. This relationship creates a hierarchy within the pattern language.
- The name of a pattern helps to refer to its central idea quickly, and build a vocabulary for communication within a team or design community.
- The ranking shows how universally valid the pattern author believes this pattern is.
- The opening illustration gives readers a quick idea of a typical example situation for the pattern, even if they are not professionals (screen shots, video sequences of an interaction, audio recordings for a voice-controlled menu, etc.)
- The problem states what the major issue is that the pattern addresses.
- The forces further elaborate the problem statement. They are aspects of the design that need to be optimized. They usually come in pairs contradicting each other.
- The examples section is the largest of each pattern. It shows existing situations in which the problem at hand can be (or has been) encountered, and how it has been solved in those situations.
- The solution generalizes from the examples a proven way to balance the forces at hand optimally for the given design context.
- The diagram supports the solution by summarizing its main idea in a graphical way, omitting any unnecessary details (UML diagram or storyboard sketch for HCI, for example)

2.5 Organizing Patterns

Connecting all patterns into a pattern language is one way of organizing them. A language can be depicted as a graph showing all pattern names and connections. However, in practice, when designers or engineers need to search for patterns for a particular problem, the graph may not be the best representation. The graph shows the fundamental relationships but there are many other practical ways in which patterns from a collection can be classified [11].

Another organizing principle is by function or problem similarity. The idea here is about grouping patterns according to their functional aspects. Certain groups of patterns may all deal with a common problem and therefore group together. Designers often need to make a decision about a functional aspect and may be best served by a set of patterns that can be classified as belonging to that functional aspect [11].

The GoF book [4] started the categorization effort by dividing patterns into “Creational”, “Structural” and “Behavioral” groupings. POSA [5] tries to take the next step by grouping patterns according to finer grained criteria such as interactive and adaptable systems, organization of work, communication and access control.

Table 2 shows a possible classification for patterns of interaction design.

Site Types	User Experiences	Navigation	Basic Interactions	Searching
My Site	Shopping	Bread crumbs	List Builder	Simple Search
Portal	Community building	Double Tab	Tabbing	Advanced Search
Commerce Site	Learning	Meta Navigation	Paging	Search Area
Corporate Site	Document Retrieval	Split Navigation	Wizard	Sitemap
News Site	Entertainment	Fly-out Menu	Sorting	Topic Pages
Branded Promo Site		Trail Menu	Enlarged Click area	Search Tips
Community Site		Image Menu		Search Index
Brochure Ware Site		Scrolling Menu		
		Guided Tour		

Table 2 – An example of a patterns structuring taken from Welie’s catalog **[11]**

However, Fowler [10] remembers that, in the end, it's more valuable to have a bunch of good patterns, poorly organized than it to have a really good structure with weak patterns underneath them.

2.6 Connecting patterns

The basic assumption in the concept of a pattern language is that patterns are related to each other, forming a network of connected patterns. These relationships are at the heart of the pattern language because they add value over single patterns. That added value is the kind of synergy we are looking for when building pattern languages.

In the patterns that are publicly available, there are already patterns that "link" to another pattern in several different ways. A closer look reveals that there are some fundamental relationships distinguishable, resembling the types of relationships known in Object Oriented Modeling. To illustrate these relationships, we will look at web design patterns as an example [11].

2.6.1 **Aggregation**

Consider the *Shopping Cart* pattern. Using this pattern, users can manage a list of items in a cart. The cart is actually a persistent list of items on which users can perform some operations such as delete, view, and change quantity. This basic behavior is covered by the *List Builder* pattern. The *Shopping Cart* is a pattern that aggregates several other patterns. This is a form of a "has-a" relationship. The *Shopping Cart* has a *List Builder* [11].

2.6.2 **Specialization**

Patterns can also be specializations of other patterns. For example, the *Advanced Search* pattern is basically a *Simple Search* but with extended options. It "inherits" the basic idea from the *Simple Search* pattern and extends it with advanced scoping, term matching and output options. We call this a "is-a" relationship, one pattern is a more specific version of another pattern [11].

2.6.3 **Association**

When designing a "shopping" experience for a particular site, there are several patterns that may also be of use. For example, when one constructs a *Product Comparison*, it is possible to offer the possibility to purchase the item directly from there, using the *Shopping Cart* pattern. This is not a "has-a" or "is-a" relationship but simple a "related-to" relationship. A pattern may be associated to other patterns because they also often occur in the larger context of the design problem, or because the patterns are alternatives for the same kind of problems [11].

2.7 Summary

In this chapter, we covered the main topics on patterns of HCI. We introduced the notions of pattern and pattern language. We described methods to identify patterns and formats to cataloging them. In the end, we gave several approaches to connect patterns.

From our analysis, it is clear that there isn't enough research on searching for interaction design patterns. They are starting to emerge as web catalogs in a collaboration environment, but they are still very few to build a full web-application only based on UI patterns.

Patterns and pattern languages offer an approach to design with much potential. It is important to note that there are researches contemplating the construction of pattern languages. However, there are very few attempts to build a complete pattern language. Existing catalogs don't give too much semantic on patterns' relationships. They describe ad hoc patterns and did not put much effort on linking them.

3 Scope of the study

In this section, we analyze various stereotypes applications that were discovered during this research to define the scope of this thesis. We start by defining applications stereotypes and explaining why they are important. We also defined layers from a typical web application and how OutSystems elements map those layers. Then, a program was developed to process several applications and output the percentages (of effort) on each layer. In the end, we conclude that patterns from *Storage and Retrieval* stereotype are the ones we want to study.

3.1 Stereotype Applications

A stereotype application is a technical solution and very high level pattern that is frequently implemented in general applications. It does not solve a business problem by itself, but it can be associated with business as part of the solution. Examples of stereotype applications are mash-ups and workflows which will get into much detail later.

These stereotype applications are Web based. It means that patterns discovered from these kinds of applications have straightforward limitations which depend on protocols (like HTTP) and very specific technology (like HTML or JavaScript).

A stereotype pattern should be easily defined from a combination of patterns. These patterns will be implemented in the OutSystems platform in a near future. Hence, we need to choose from a set of patterns the ones that bring higher value to the platform in terms of applicability, frequency of use, productivity enhancement and highly customization to support built-to-change demands from business.

The reason why we are studying stereotypes is because we need to narrow our vision and concentrate on patterns that bring higher returns. To achieve such goal we analyze applications that correspond to a stereotype and we try to extract the effort made by its developers. That is why it is important to find patterns that reduce those efforts.

3.2 Stereotypes Enumeration

The stereotype applications identified are:

- **Forum** – application for holding discussions and posting user generated content. The configuration and records of posts can be stored in text files or in a database. Each package offers different features, from the most basic, providing text-only postings, to more advanced packages, offering multimedia support and formatting code (usually known as BBCode).

- **Storage and Retrieval (CRUD)** – deals with CRUD actions around entities: create or add new entries; read, retrieve or view existing entries; update or edit existing entries; and delete existing entries. A CRUD application uses forms to get data into and out of a database. These operations are often documented and described under one comprehensive heading, such as "contact management" or "contact maintenance".

- **Integration Broker** – enables diverse applications to exchange information in dissimilar forms by handling the processing required for the information to arrive in the right place and in the correct format. In addition, a broker may facilitate the application of user-defined rules or business logic to the processing of the data. Data exchange is performed by the integration broker without requiring applications to have any knowledge of the data conventions or requirements of the applications receiving their data. It usually contains an engine so that external systems can subscribe as providers of data or messages. This engine allows defining the flow of such data or messages into other systems. Thus, it is used for marshaling/un-marshaling to/from an intermediate format (like txt, csv, xls, xml, etc).

- **Aggregate and View (Mash-up)** – combines data from more than one source into a single integrated tool (mash-up) and then creates a new view of the data. A mash-up application is architecturally comprised of three different participants that are logically and physically disjoint:

    - API/content providers to expose content through Web-protocols such as REST, Web Services or RSS

    - Mash-up site where mash-up logic resides and it is not necessarily where it is executed

    - Client's Web browser to assemble and compose the mashed content.

- **Analytics (Reports and Charts)** – in businesses and enterprises of all kinds, organizing and presenting information has traditionally been the job of documents called reports. These documents generally consist of multiple pages that can include text, numbers, charts, maps, and illustrations. The best reports convey the facts needed to make the best decisions, unobscured by a clutter of data irrelevant to the task at hand. Analytics are reports with business logic. Common features includes printable version, agglomeration, customizable reports for end-users, reporting of charts, data mining, data warehousing and cubes.

- **Batch Processing** – an application based on the execution of a series of programs ("jobs") on a computer without human interaction, so all input data is preselected through scripts or command line parameters. It avoids idling the computing resources with minute-by-minute human interaction and supervision. If that’s the case, it shifts the time of job processing to when the computing

resources are less busy. Batch Processing may be used within asynchronous processes for gathering data (outside the system) from time to time, to send notifications to the user, etc.

- **Workflow** - a sequential (and fixed) workflow represents a workflow as a procession of steps that execute in order until the last activity completes.
- **Document Storage** - application where documents are stored in the File System (instead of a DB). It has some characteristics such as: versioning to identify the last one and allow navigation through its history; searching for documents and its contents; and access and control permissions.
- **Content Management System (CMS)** – is used to collect, manage, and publish content, storing the content either as components or whole documents, while maintaining dynamic links between components. A wiki, for example, is a CMS which enables documents to be written collaboratively, in a simple markup language using a web browser. It allows users to easily create, edit and link web pages collaboratively.

3.2.1 Orthogonal Stereotypes

Orthogonal Stereotypes are stereotypes that can actually be applied over almost all the other stereotypes. In this sense, they are more patterns and less technical application stereotypes or a stereotype property. As they can be quite high level and materialized in different fashions, they appear here as stereotypes.

We identified two orthogonal stereotypes:

- **Web Collaboration** – provides an organization with the capability to collaborate with stakeholders (employees, customers, suppliers, etc). Web collaboration provides features such as: tagging, blogging, comment or ranking. It can use rich Internet application techniques (often Ajax-based), offering: voice and text chat assistance or to conduct single or multi-user conferences and seminars, bidding system or voting systems.
- **Runtime Application Building Engine** - applications rich in runtime behavior like the ability to define rules, queries, processes, formulas in runtime. IT defines a set of tasks and afterwards business users have the ability to define processes composed with those tasks. In Salesforce.com, the user can save filters (queries) and then use them later.

3.3 Validation and Business Value

These application stereotypes were identified and validated with other stakeholders (project managers and senior consultants) from OutSystems. This validation was conducted with individual meetings and we came up with the solution presented in this thesis.

We are now able to define an application based on these stereotypes. For example, we can say that the functionalities of an application X are distributed along with the stereotypes: 30% of the features belong stereotype Y, 60% belong to stereotype Z, and so on. In other words, it is now possible to catalog applications in terms of stereotypes. This is useful for Sales department from OutSystems who easily create new budgets based on the catalog.

Another important issue from this stereotype analysis is that we named our contexts where we operate in. In certain contexts it makes sense use certain patterns.

However, in this report, we are not getting into much detail with this cataloging, because it is not important for our final discussion and there is no need to lose our path. We are just considering that an application is mainly composed by a single stereotype. This is really useful because it allows us to study just two or three stereotype applications to find all the majority of patterns we need.

3.4 Defining Layers and Classifying elements

OutSystems applications are defined by an OML (OutSystems Mark-up Language) file. This OML contains a set of annotations XML-like to describe a web application. It has information about screens, widgets, buttons, actions, business entities, styles, etc.

We define four layers in an OML file:

- *User Interface (UI) Layer* – Handles presentation elements to display information to the user [37]. Presentation elements are displayed on screens (hold the complete definition of the visual content as a collection of elements) and screen flows (set of screens liked through connectors). We can elements like forms, buttons, links, tables, widgets, JavaScript, CSS and even images.
- *Business Logic (BL) Layer* – Implements the behavior of the application by performing detailed processing, specific to the business domain [37]. The element types in the BL layer are:
    - Action – implements a specific behavior. There also exist built-in or user-defined actions. Examples of built-in actions are: *Login*, *Logout*, and *Commit Transaction*. User-defined actions are graphically coded using a specific editor provided by OutSystems Platform. We can find nodes like *if*, *assign*, *switch* and *foreach*.

    - Variable - writable element in the scope of a screen or of a action flow that holds non-persistent data
    - Parameter – used exchange information between screens and actions
- *Data Layer (DL)* – Manages and structures data. It keeps data neutral and independent from application servers or business logic [37]. Elements on DL are, for example, entities (persistent structured information), attributes (part of information that concerns the entity), queries, queries parameters, entity CRUD actions and foreign keys attributes.
- *Third-Party Technologies (TT) Layer* – Deals with all the non-OutSystems technologies used when building an application in all layers. Examples are embedded HTML, JavaScript, web services and foreign entities.

We mapped each element from the *OutSystems Platform* to each layer above. As there are a lot of elements in the language (around 110) with unnecessary details, we are not going to display the resulting table. Instead we display an image to clearly illustrate the overall mapping, as it resumes what was written above.

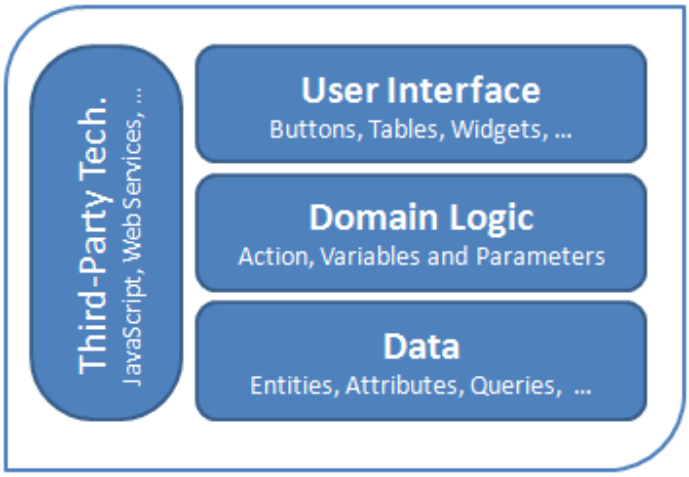


Figure 1 – Layers and mapping from OutSystems Platform (adapted from [37])

3.5 Results

To turn these results trust-worthier, we chose applications which fully correspond to a stereotype. As we have already mapped the elements with layers and applications with stereotypes, we can now analyze the applications and interpret the results.

Stereotype	OML from App.	UI	BL	DL	TT
Forum	OS Community Forum	77%	14%	9%	74%
Storage and Retrieval	Supplier Self Service	68%	19&	13%	62%
Integration Broker	SSS SAP MM	4%	79%	17%	28%
Workflow	Issue Manager	38%	43%	20%	27%
Document Storage	File Storage	45%	43%	10%	45%
CMS	OS Wiki	65%	28%	7%	59%
Web Collaboration	SSS Collaboration Serv.	91%	8%	1%	89%

Table 3 – Table of percentages of elements from layers in terms of stereotypes

Firstly, let's consider the number of elements in an OML is proportional to the effort of developing the application. It means that the higher the percentage in a layer, the higher the effort of developing the application. Another important fact is that 80% of OutSystems' applications correspond to *Storage and Retrieval* stereotype.

As we can see from the table 3, the biggest effort, from each layer, is generically concentrated in *User Interface*. It means that if we implement patterns that can accelerate development processes, we get an higher return in terms of business vale.

In this study, there is not enough time to study all the patterns from all the stereotypes discovered. So, we chose the *Storage and Retrieval* stereotype because OutSystems can get higher productivity if patterns from the most popular stereotype are implemented.

Interaction design patterns are the ones we need to catalog and implement because they dramatically help in enhancing productivity and supporting built-to-change requirements from everyday businesses.

3.6 Conclusions

In this chapter we discovered several high-level patterns, or technical application stereotypes, that help us to define which kinds of solutions we need to consider. Otherwise we would end up with searching for all applications, lost on identifying patterns like every public catalog.

Each stereotype has a set of common patterns. We defined layers from where elements can be belong to and analyzed the stereotype applications in terms of those layers. We concluded that studying patterns of interaction design, from *Storage and Retrieval* stereotype, are the key for higher returns, in terms of productivity enhancement.

4 Pattern-Language Definition

The purpose of this section is to provide a set of patterns that defines the structure of Web 2.0 storage and retrieval applications. By composing patterns over patterns, we can generate the whole application, without configuring a huge set of parameters, nor developing each pattern individually. The patterns specified are the ones found in enterprise applications that are storage and retrieval based. There is also an incentive for specify built-to-change patterns, because enterprise applications need to meet the demands of continuous changing business.

It is important to emphasize that we do not bother with how patterns will be instantiated, neither with which tools they will be implemented, nor with classifying them at first.

In section 4.1, we describe the process for mining patterns, from their identification to their validation. In section 4.2 we explain how patterns are composed by other patterns and afterwards we present our pattern-language graph (section 4.3). Finally, in section 4.4, we introduce the format to completely define a pattern.

4.1 Pattern Mining Process

In section 3, we concluded that interaction design patterns or UI patterns are the ones we are trying to find. An interaction design pattern is a general repeatable solution to a commonly-occurring usability problem in interface design or interaction design.

In this process we searched for two popular applications rich in *Storage and Retrieval* patterns. We then proceeded with the identification and respective validation.

4.1.1 Applications Overview

As we knew that our first reference was Storage and Retrieval patterns, we searched for applications which correspond to the stereotype. We studied in deep, three main applications:

- Salesforce.com – a very successful easy-to-use Web-based CRM solution for sales, service, marketing, and call center operations that streamlines customer relationship management and boosts customer satisfaction.
- Supplier Self Service - is a web-based turnkey solution that assists procurement organizations and suppliers to collaborate and manage their daily interactions in real-time, across the entire procure-to-pay process.

Today, storage and retrieval (or data-centric) applications are the predominant kind of application found on the web [38]. Storage and retrieval applications rely on a connection to a database where the bulk of its processing involves querying a database and returning results. The information is represented by a collection of data abstracted from observations of the real world and made available to the system. A person uses an information system in two major ways: to store information in anticipation of a future need and to find information in response to a current need. In either case, the user has some information need that drives the use of the information system [39].

In these types of applications, like Salesforce (CRM application) [40], users manipulate a set of core entities supported by a set of look-ups. There are a set of core entities which have their place in the navigation menu (like Leads, Accounts, Opportunities, etc.). When users access a core entity (by clicking on the menu, for example), they can manipulate it by inserting new items, editing them, viewing them or even deleting them. It is also possible to manipulate secondary entities, like attachments or comments, which have a strong connection with the core entity. Look-ups are also very common in the attributes of an entity. They often categorize an item or define its state.

There are different users of these applications with different responsibilities. Some users can only access some entities, while others can have a larger view of the application. An administrator can even have a back-office where he can manage users (and their roles) and potential look-ups used throughout the application.

4.1.2 **Patterns Identification**

Patterns are usually discovered by reference to design solutions, rather than being constructed from first principles. In the very beginning, we start exploring applications by observation, and found patterns that we were unable to sort or apply any semantic relationship between them. We identified about 450 common patterns from lower granularity (e.g. *authentication pattern* – with features like *login*, *logout*, *signup*, *forget password*, etc.) to high granularity (e.g. *input form validation*).

Another important thing we face on the beginning was the naming of patterns. We had patterns which a big nomenclature such as: *Set Maximum Invalid Login Attempts and Lockout Period*, *Set Number of Table Record Lines in Runtime*, *Persist List Records Filter Through Sessions* or *Process Feedback Monitoring Upon End-User Action*. Identify the “quality without a name” is easier than giving it a name. Naming patterns is a hard work and, by the time we are delivering this study, we are still renaming them. However, naming patterns does worth it. It gives us a standard nomenclature, a lingua franca, where everyone can refer to a pattern unambiguously.

Additionally, in case the name was not (obvious) enough to quickly identify a pattern we added a little description of the pattern and links to the web pages that references them.

It is important to note that in the beginning we grouped them in folders (to achieve a kind of hierarchy between them), because it was really impossible to manage such a huge number of patterns without any kind of organization between them.

In the end, we start organizing them into a graph to come near with what we have in mind: developing a full web application only based on patterns. We will come back into this subject on section 4.2 where we explain how they connect to each other.

Later we adopted a specific pattern format that clearly specifies the way we need to define a pattern. Again, we will get into it on section 4.4.

4.1.3 **Patterns Validation**

Pattern mining starts with identification of good practice. However, many interaction design patterns can be criticized for identifying common rather than necessarily good practice. Hence, we recur to professional services, from OutSystems, to validate that those patterns are the ones that bring higher values.

We started by prioritizing all the patterns by frequency as they appear in the applications we chose. We selected the patterns with higher frequency (about 100 patterns) among the applications and dropped the rest. We then schedule three interviews with three different consultants (one consultant at a time), we ask them to prioritize the patterns with *Must Have*, *Should Have* and *Nice to Have*. As they work every day in a customer facing function, they know which patterns (if implemented) would bring less effort on implementing newer applications.

4.2 Connecting Patterns and Patterns Composition

Just as words must have grammatical and semantic relationships to each other in order to make a spoken language useful, design patterns must be related to each other in order to form a pattern-language. Our pattern-language is defined hierarchically where the instantiation of patterns in the domain depends on the current context (or pattern). This means that instantiation of patterns may require previous instantiation of others. For example, we can only use the *Pagination* pattern in the context of an *Entity List*. However, an *Entity List* doesn't need a Pagination pattern to be instantiated. Hence, our pattern format supports the definition of connections between patterns, in order to define and support a pattern-language.

In our pattern-language, based on a graph, patterns are composed and instantiated depending on other patterns. Then it is necessary to define a set of rules that allows pattern composition. The pattern's format must clearly reflect the conditions where and when a pattern must and can be instantiated.

From a developer point of view, a pattern is defined by a set of inputs and customization points. Inputs are mandatory information that the pattern needs to be instantiated. Customization points are characteristics of the pattern that can be changed, or operations the developer can easily make. For example, an *Entity List* has a “data source variable” (or a record list) as input, and “alternating row color” as a customization point.

Unlike common patterns, these patterns follow a paradigm of convention over configuration. This means each pattern must have a default representation associated, which is a frequently use combination of pre-determined customization points. For example, we can apply *Pagination* over an *Entity List* and the number of records shown is 20. As 20 can be a good or bad number depending on the context, it can be modified if the developer feels the need to. But the idea is to provide the developer with maximum defaults, thus requesting minimum information to provide greater productivity.

When we compose patterns in a graph, a set of mandatory inputs and customization points may be imposed from parent patterns to children patterns. Inputs and customization points are needed not only to share information about development dependencies, but also to connect the pattern's flow. One parent pattern can fill in the inputs (and customization points) of one of its children. For example, if we are defining an *Edit Form* and set red color in a customization point, the different *Edit Form Fields* that compose the pattern will automatically have red color set in their customization points. However, we can drill down to each *Edit Form Field* and customize it to define a different color, if we need to.

4.3 Patterns-Language Graph

To illustrate how patterns are composed with other patterns we present our huge graph of patterns, in the next figures.

Note that in order to define first the more important patterns (which are the ones that bring higher returns in terms of productivity enhancement), we divided them in “MUST Have”, “SHOULD Have” and “NICE to Have”. An aggregator is just a way of aggregating patterns in something with a name in order to refer to the set instead of referring to all the patterns included in the aggregator.

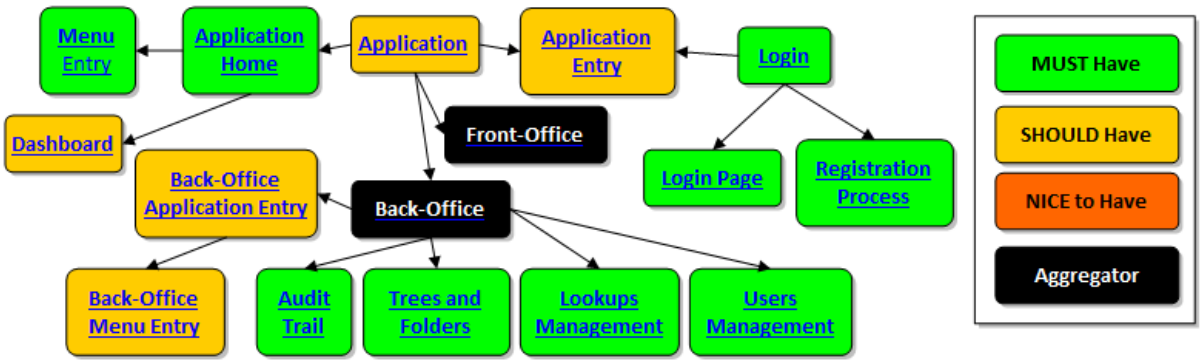


Figure 2 – Patterns Graph with focus on *Application* pattern

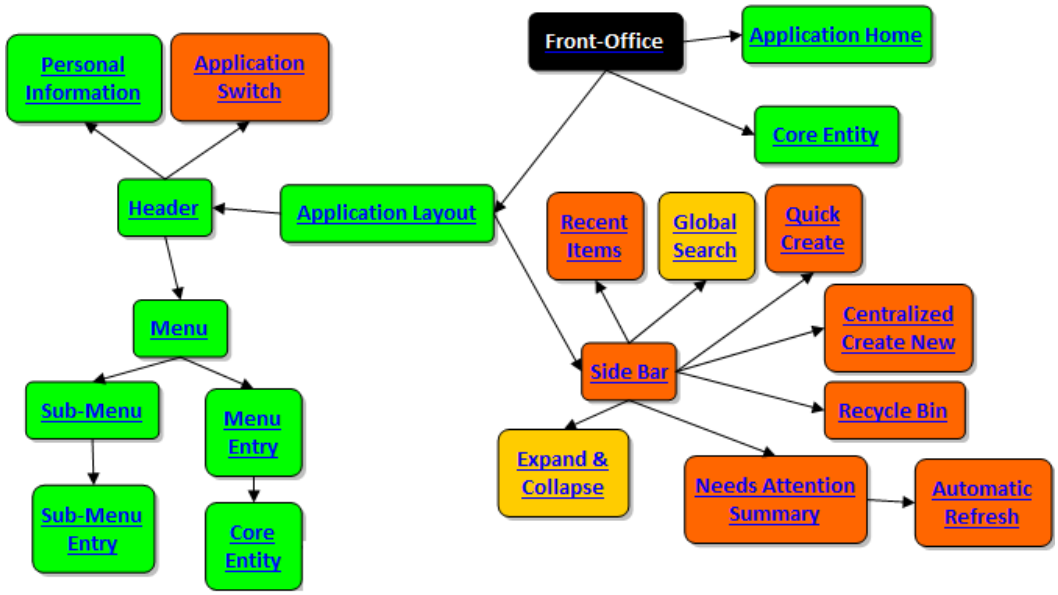


Figure 3 – Patterns Graph with focus on *Application Layout* pattern

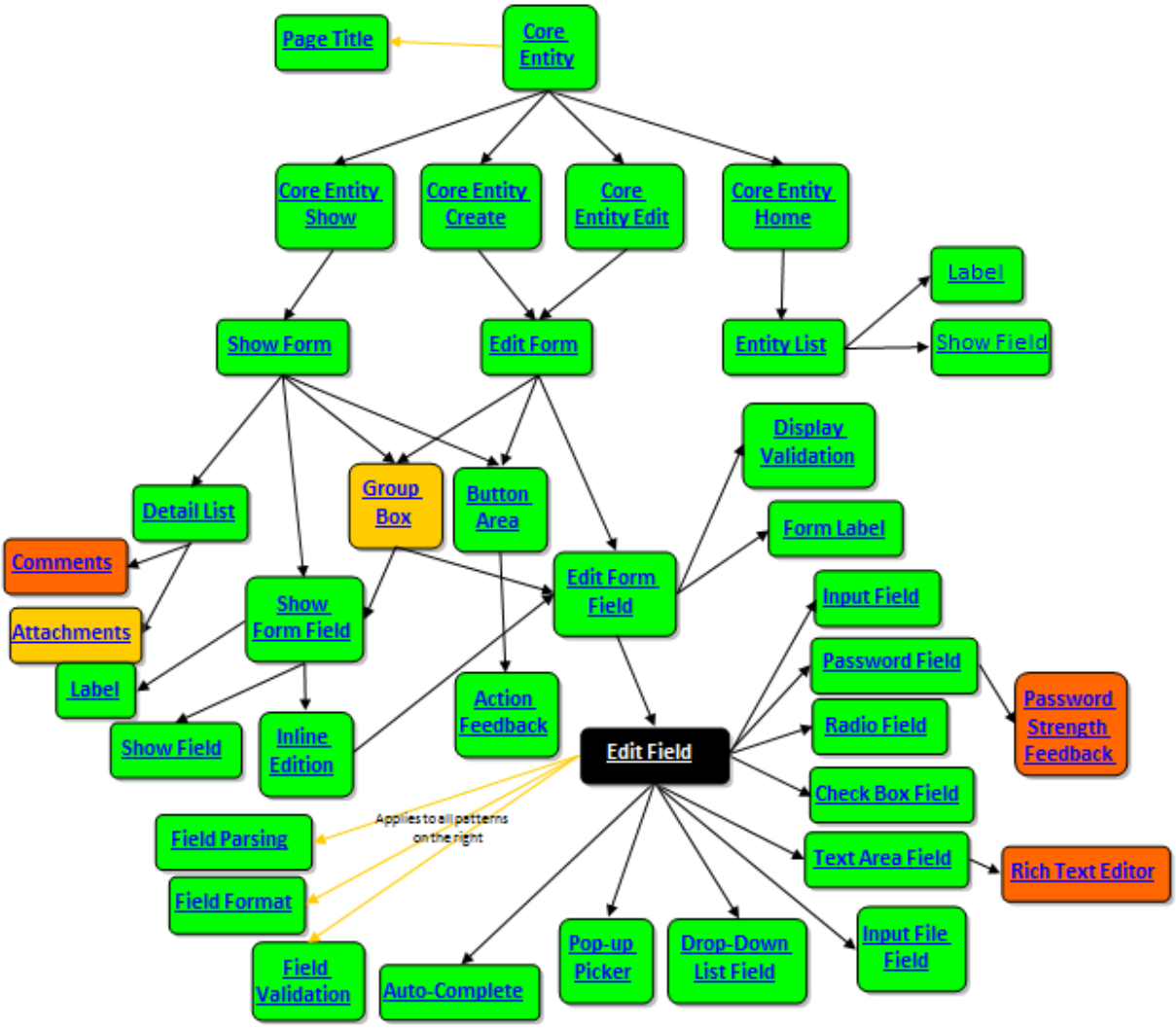


Figure 4 – Patterns Graph with focus *Show Form* and *Edit Form* patterns

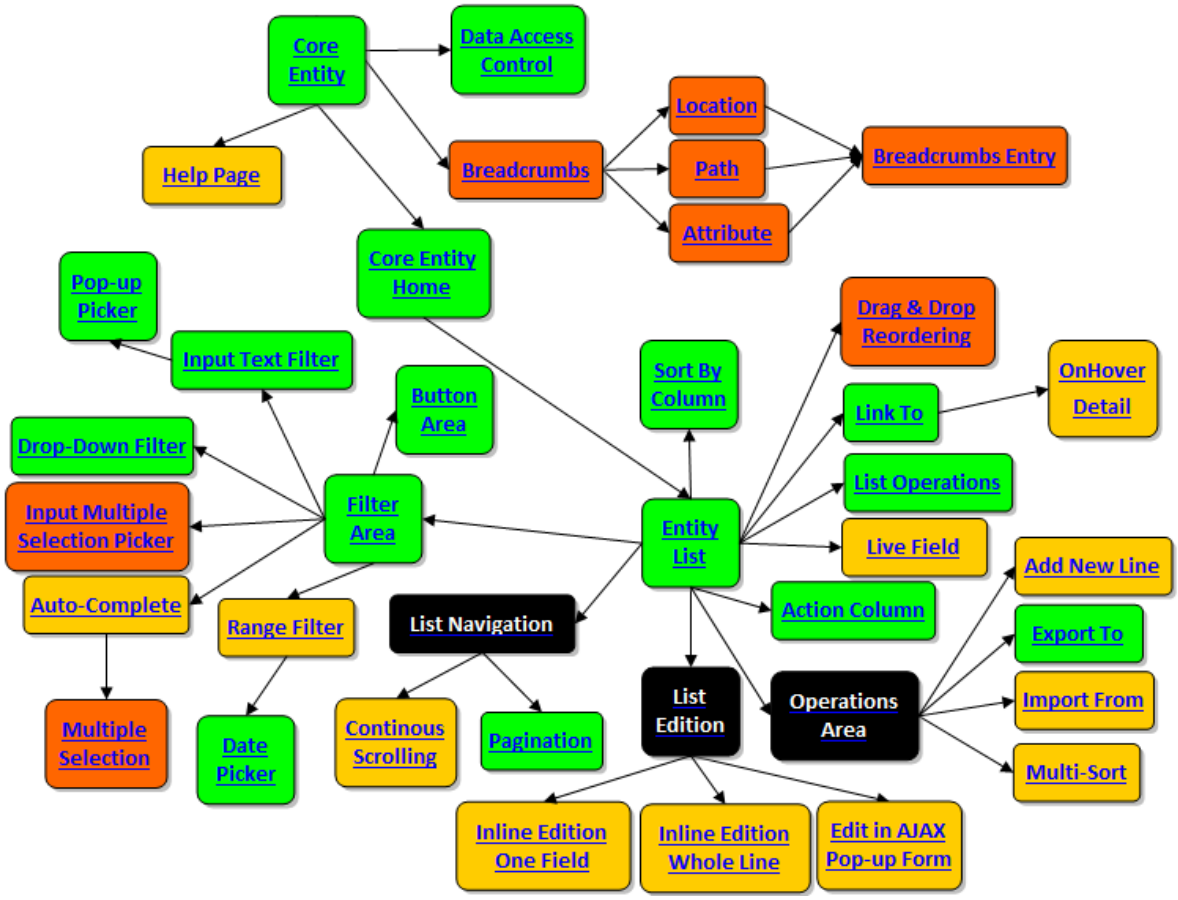


Figure 5 – Patterns Graph with focus *Entity List* pattern

4.4 Patterns Format

This study about patterns and relations between them, proposes a pattern relation model, to formalize the semantic structure of a pattern language. We also use this model to facilitate the development of a full web application that will describe on chapter 5.

In section 2, we showed lots of different ways we can write our patterns. Every book typically uses its own, and many papers show several variations. There also appears to be a lack of consensus about the format and focus on interaction design patterns. Consequently, a pattern language for interaction design has not been established since it is necessarily preceded by the development of a sufficiently large body of patterns with the same forces or format.

One thing is describing recurring patterns; another is how to actually implement the patterns. In this format we don't want to describe the techniques to implement a pattern, but to provide the developer with the information required for implementation. We can think of this information as the one needed to implement a design pattern. As must patterns use a user perspective, the developer perspective is becoming extinct or very dependent on the technology. In this study, the format considers inputs and customization points that the developer of the pattern may consider while implementing the pattern, so that he can latter reuse the implemented pattern.

Much of domain knowledge has been captured in well established pattern language. The following format tends to be platform agnostic by defining the relationships between patterns, inputs and customization points. As explained before there are two views: user and developer. The user view focuses on usability and why the pattern is a good solution for the problem in terms of end-user. The developer view responds to the pattern composition issues.

The following tables explain each field of the pattern format we chose to formally define a single pattern.

Problem	The problem is related to the usage of the system and is relevant to the user or any other stakeholder that is interested in usability and functionality. The problem description should often be user task oriented.
Solution	A solution must be described very concretely and must not impose new problems. However, a solution describes only the core of the solution for the problem and other patterns might be needed to solve sub-problems.
Images	The images should picture how the solution has been used successfully in a system that solves the problem. An example can often be given using a screenshot and some additional text to explain the context of the particular solution. It is preferable to use examples from real-life systems so that the validity of the pattern is enforced, if possible.
Use When	Situations (in terms of tasks, users and context of use) giving rise to a usability problem. This section extends the plain problem-solutions dichotomy by describing situations in which the problems occur and the pattern is a good solution.
Use Cases	Describes the interaction use cases with the pattern and the returned behavior of the pattern. Use cases are sequences of simple steps from a certain perspective (actor). It can include images.

Why/Rationale	This section describes why the pattern works and why it is good. The solution section describes the visible structure and behavior of the pattern, while the rationale provides insight into the deep structure and key mechanisms that lie under the surface of the pattern. The rationale should describe which usability aspects are improved or which other aspects make this solution be a good one for the problem. Other parallel patterns relevant to the solution should be referenced.
Requirements	A condition, constraint or capability that must be met or possessed by the pattern to satisfy an imposed specification.

Table 4 – OutSystems Form: End-user view

Mandatory Inputs	Inputs that are mandatory to provide the pattern with. Like a programming function, a pattern may have inputs that need to be fulfilled in order to be instantiated.
Customization Points	Operations and/or characteristics that can change the behavior or look of the pattern without losing its identity. Like alignments, colors, name of some text, etc.
Is Part Of	Patterns that depend on this pattern to exist. Without this pattern, parent patterns cannot exist. If some child pattern inputs are defined by the parent pattern, the parent pattern has the possibility to lock inputs and customization points (if it is necessary).
Is Used By	Patterns that can use this pattern but don't depend on it to exist.
Composed By	Patterns that must be implemented in order to instantiate this pattern. Without "composed by" patterns, this pattern cannot exist. When a pattern (parent) is defined by other patterns (children), the parent pattern can customize the child patterns by defining their inputs and customization points (if it is necessary). If a parent pattern doesn't define a child pattern input, the responsibility of filling the input stays with the child pattern.
Can use	Pattern that this pattern can use to enrich itself. Referencing patterns in this field means that the current pattern doesn't need these child patterns instantiated to exist. These patterns can be seen as a decoration to the current pattern.

Table 5 – OutSystems Form: Developer view

5 Patterns Specification

About 50 patterns were fully described using the form from previous chapter. However, we are not going to present them all in this section because the idea is to give an overview of the pattern-language and not an exhaustive description. We only describe the patterns that we were needed in the next chapter.

5.1 Application Layout

Problem	The user needs to have the content and functionality structured and organized in the application
Solution	The webpage is divided in sections where each one has its own responsibility
Images	(Appendix B)
Use When	Always. In enterprise web applications there is always a section which serves the main navigation and another section for displaying content.
Use Cases	User looks at the layout and realizes where is the navigation area and the dynamic content area
Why/Rationale	The idea of an application layout is to provide the structure of a page.
M. Inputs	-
C.Points	Position of the *Side Bar* Position of the *Screen Content*
Is Part Of	*Front Office*
Is Used By	-
Composed By	*Header* *Webpage Content*

Can use	*Application Entry* *Footer* *Page Title* *Side Bar*

Table 6 –*Application Layout* pattern form

The image below shows the *Application Layout* pattern where we can instantiate patterns like a *Header*, *Footer* and *Side bar* patterns.

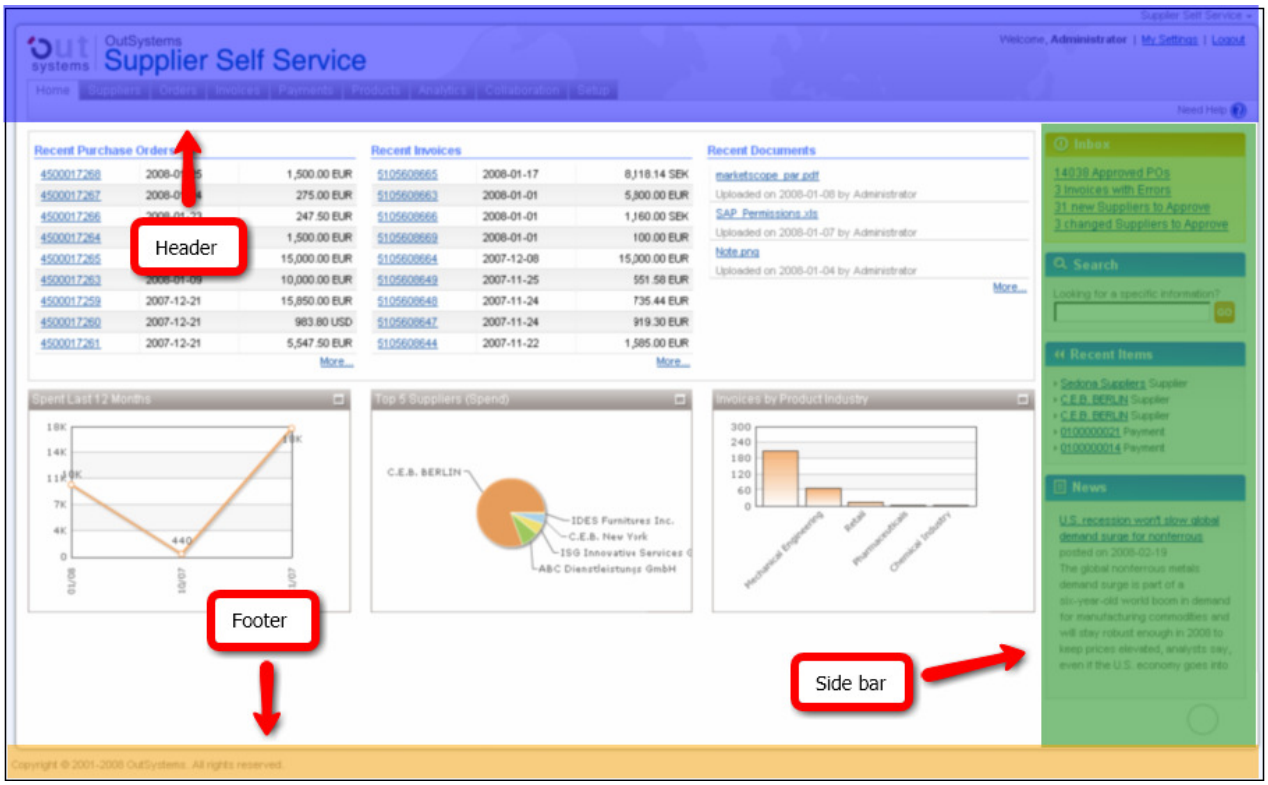


Figure 6 – *Application Layout* composed with *Header*, *Footer* and *Side bar* patterns

5.2 Core Entity / CRUD Pattern

Problem	The user needs to manage a set of items. Typically he wants to create, read, update and delete (CRUD) them.
Solution	A sequence of screens that allows the user to create, read, update, and delete the items belonging to a core entity
Images	-
Use When	Whenever there is a need to manipulate an entity
Use Cases	1. User sees a list of items 2. After scanning the list, user clicks on a link (to *Show Form* pattern) of an item/row 3. User lands on a *Show Form* pattern corresponding to that particular item 4. User decides to edit the content and clicks on Edit button in *Button Area* 5. User lands on a *Edit Form,* edits a field and clicks on Save 6. User lands on a *Show Form* and confirms that the change was successfully made 7. He clicks backs to the *Core Entity Home*
Why/Rationale	The objective of this pattern is to generate a skeleton in order to increase productivity. It is supposed that the user customizes this pattern after its generation.
Requirements	Scaffolding the different Core Entities.
M. Inputs	Entity
C.Points	Style
Is Part Of	-
Is Used By	*Menu Entry*
Composed By	*Core Entity Show* (Entity) *Core Entity Create* (Entity)

	Core Entity Edit (Entity) *Core Entity Home* (Entity)
Can use	*Data Security*

Table 7 – *Core Entity* pattern form

5.3 Entity List

Problem	Users need to see the items stored.
Solution	Once the items have several characteristics the solution is a table where: each item is represented in a row, and each column represents each item's characteristic.
Images	Appendix C
Use When	User wants to display the list of items.
Use Cases	1. User scans over the *Entity List* 2. User doesn't find the item he wants and clicks on page 2 3. Page 2 is displayed 4. User sorts (alphabetical ascending) the column Supplier and still doesn't find what he is looking for 5. He knows that the item have is from business unit ABC 6. He selects that business unit in the filter and clicks on "Apply Filter" 7. He founds the item and clicks on the link to *Show Form*
Why/Rationale	Entity List serves the need to organize data belonging to the same entity.
Requirements	Differentiate the column headers from each row On hover highlight for each line
M. Inputs	Source Record List Variable
C.Points	List Title

	Visible columns Order of columns "OnEvent" operation Alternating row color Round Corners Manual order by Links to Columns Width
Is Part Of	-
Is Used By	*Core Entity Home* *Detail List*
Composed By	Pair *Label - Show Field*
Can use	*List Navigation* (Entity) *Filter Area* (Entity) *Sort By Column* (Entity) *List Operations* (Entity) *Action Column* (Entity) *Operations Area* (Entity) *List Edition* (Entity) *Drag & Drop reordering* (Entity) *Live Field* (Entity)

Table 8 – *Entity List* pattern form

The following image show a very simple *Entity List* pattern. Please check Appendix C so see a more complex one.

Name	Email	Creation Date
Albert Einstein	albert@outsystems.com	2008-01-01 00:00:00
Luís João	luis.joao@outsystems.com	2008-01-01 00:00:00
Alfred Hitchcock	alfred.hitchcock@outsystems.com	2008-01-01 00:00:00
Jason Statham	jason.statham@outsystems.com	2008-01-01 00:00:00
Alan Ford	alan.ford@outsystems.com	2008-01-01 00:00:00
Meryl Streep	meryl.streep@outsystems.com	2008-01-01 00:00:00
Elizabeth Taylor	elizabeth.taylor@outsystems.com	2008-01-01 00:00:00
Jodie Foster	jodie.foster@outsystems.com	2008-01-01 00:00:00
Penelope Cruz	penelope.cruz@outsystems.com	2008-01-01 00:00:00

Figure 7 – A very simple *Entity List* pattern built with OutSystems Platform

5.4 Master/Detail

Problem	The user needs to work with different sets of information units linked by a relationship.
Solution	The main information unit (the master) determines the slave information units (details). An Entity List (with a title), linked with a certain record. This list can appear behind the Show Form or in tabs.
Images	Appendix D
Use When	Applications that aggregate several objects (with relationships between them) and the user needs to interact with all of them. The master and detail must have some kind of relationship (1-1, 1-many, many-to-many).
Use Cases	1. User views a master record (of type Account) as *Show Form* 2. Behind the *Show Form* there is one *Detail List* with contacts 3. User adds a new contact by pressing "Add Contact" 4. After adding, the contact is connected with its respective account
Why/Rationale	The joint presentation of two or more information units allows the user to work in a unique scenario capable of performing several tasks and, at the same time, the scenario maintains the details synchronized with the respect to the master component.

	Inline edition makes the edition easier without changing the context.
Requirements	Clear distinguishing between master and details. Ease changing in both master and details.
M. Inputs	Source Record Variable Entity
C.Points	Positioning Expression in Title Style
Is Part Of	-
Is Used By	*Show Form*
Composed By	*Entity List* (Entity, Title)
Can use	*Button Area* (New)

Table 9 – *Master/Detail* pattern form

Figure 8 – *Master-Detail* pattern from Salesforce.com

5.5 Edit Form

Problem	The users want to edit one or more attributes of an entity
Solution	A set of *Edit Field Forms*, each one mapping one single attribute of an entity and a *Button Area*
Images	Appendix E
Use When	The users want to edit one or more attributes of an entity
Use Cases	1. User lands in a page containing an *Edit Form* 2. User changes one or more inputs in *Edit Field Forms* 3. User saves the changes and return to *Show Form*
Why/Rationale	Using only *Edit Form Fields* is appropriate to multiple changes at the same time. Two *Button Areas* above and below the form is appropriate if the form has too many form fields.
Requirements	Edit Field Form alignment. Default Edit Form Fields are generated depending on the attributes of the given record.
M. Inputs	Flag edit/create Source Record Variable
C.Points	Data Source Number of columns Alignment of *Edit Field Forms* Order of *Edit Field Forms* Number of Button Areas (1 or 2) Style

Is Part Of	*Core Entity Edit* *Core Entity Create*
Is Used By	-
Composed By	*Edit Field Form* (Field) (1..*) *Button Area* (1..2)
Can use	*Group Box* (Fields)

Table 10 – *Edit Form* pattern form

The following figure represents an *Edit Form* composed by *Button Areas* and *Group Boxes and EFF*.

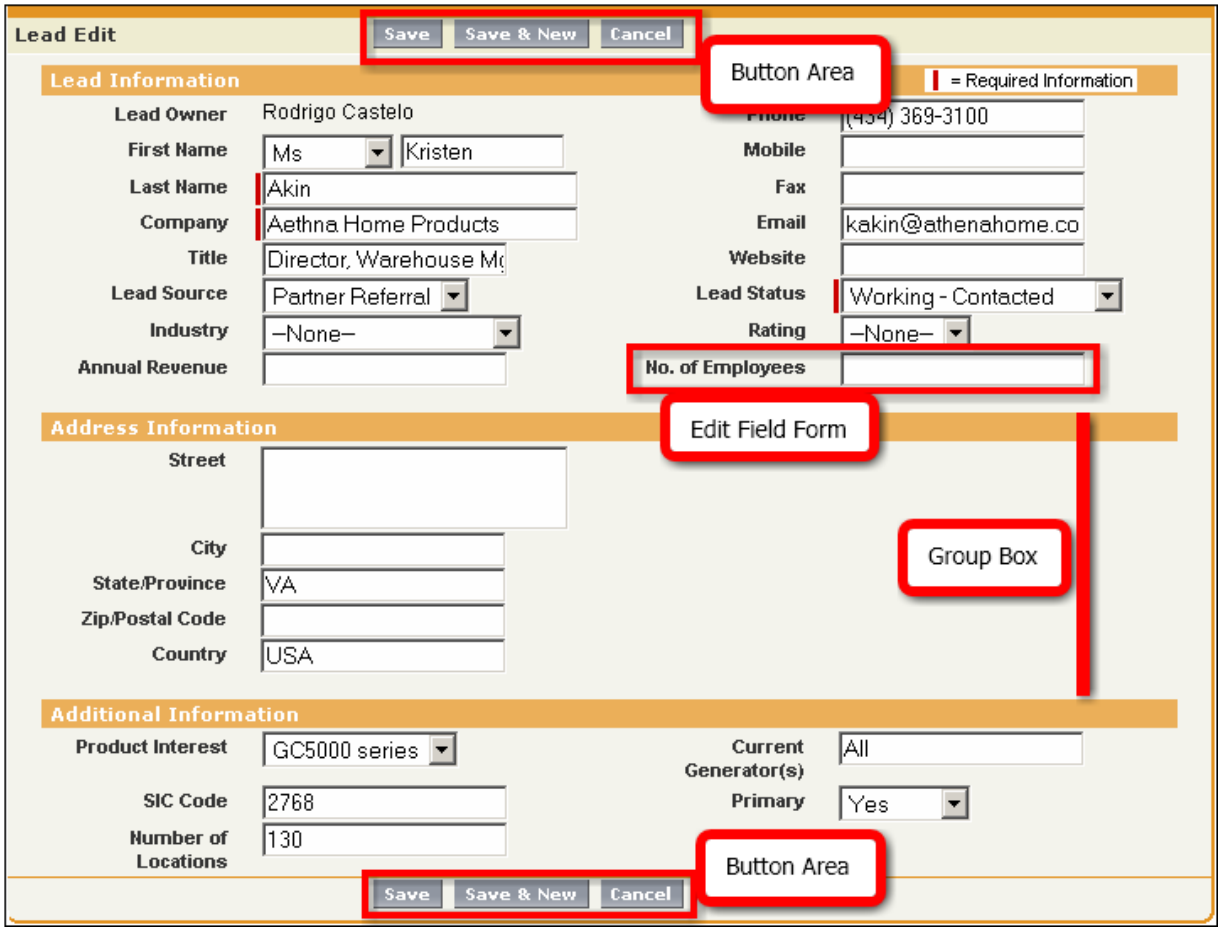


Figure 9 – *Edit Form* pattern (1/2) from Salesforce.com

5.6 Edit Form Field

Problem	The user needs to edit one attribute of an entity
Solution	A pair label-input. *Label* is a text typically corresponding to the name of the attribute (*Label*). The input is materialized as an input html tag of type: radio, checkbox, text, password or file (*Edit Field*). Positioning Children Patterns: *Label* can be above or before the *Edit Field*.
Images	Appendix F
Use When	In the context of an *Edit Form.*
Use Cases	1. When looking at *Edit Form*, the user is searching for the label that matches the attribute he/she wants to edit 2. He/She clicks in the label and the cursor positions itself in the inbox 3. He/She changes the input content
Why/Rationale	Putting the label on the right is easier to read and to search for the label we want. According to Fitts's law, using clicked label (<label> tag) reduces the time required to put the cursor into the input.
Requirements	Information about whether or not the field is mandatory
M. Inputs	Source Record Variable
C.Points	Visibility Positioning Children Patterns Style Mandatory Style
Is Part Of	*Edit Form*
Is Used By	*Group Box*
Composed By	*Label* (Text)

	Edit Field (Field)
Can use	*Display Validation* *Pop-up Picker*

Table 11 – *Edit Form Field* pattern form

Figure 10 – *Edit Field Form* pattern with a mandatory field and and input field

5.7 Filter (Area)

Problem	User needs to find an item or a set of items in a table of items
Solution	Dedicated area above the *Entity List* A mechanism to query the items satisfying certain conditions. There are various types of filters: • Single-attribute filter 1. *Drop-down Filter* - if there are less than 20 different items 2. *Input Multiple Selection Picker* - if there are 20 or more different items 3. *Input Filter* with Pop-up picker - if there are more than 20 items 4. *Auto Complete* 5. *Range Filter* • Multi-attribute filter 1. *Input Filter* Two buttons in *Button Area*: • Apply Filter/Search - to retrieve a filtered *Entity List* • Reset - to retrieve a cleaned *Entity List*
Images	Apendix G
Use When	User knows the name or part of the name of a characteristic of an item.

Use Cases	1. Simple Search 2. User types part of the name of the item 3. Clicks on search button 4. The list of items that match that name is displayed
Why/Rationale	Filtering allows users to quickly reduce the amount of items shown and help them to adjust their information to the task. For expert users, multi-dimensional filtering on all columns can be a very powerful feature that can replace reporting functionality.
Requirements	Button to reset the filter
M. Inputs	Searchable attributes
C.Points	Searchable attribute(s) Searchable column(s) Range attributes Stateless Multi-Attribute Filter Single-Attribute Filter Lookup Expand Collapse
Is Part Of	*Entity List*
Is Used By	-
Composed By	-
Can use	*Button Area* *Input Text Filter* *Range Filter*

	Drop-down Filter *Pop-up Picker* *Auto Complete*

Table 12 – *Filter* pattern form

The following figure presents a *Filter* pattern applied to the top of an *Entity List* pattern.

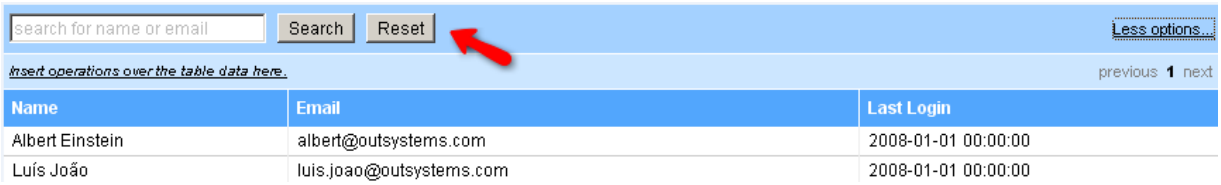


Figure 11 – Simple *Filter* pattern

6 Evaluation

The purpose of this chapter is to test the framework by building an application using the pattern language defined in the previous chapter. We go through the visual development of a simple web application using pattern-driven development.

Let's take a simple storage and retrieval example like an Account and Contact List application, where the user can add, delete, edit and view the list of contacts and accounts. The contact always belongs to only one Account. Entity Account contains many Contacts.

Let's imagine that we have a framework for designing web applications that only have interaction design patterns as elements. This means that the developer must implement the whole application based on those patterns which are provided by the framework.

The first pattern we are going to instantiate is the *Application Layout* pattern. It is composed by four other patterns:

- *Header* - static element of the application, its first purpose is to describe the navigation
- *Page Title* - area reserved for the title of the content and support specific navigation by displaying links below
- *Web Page Content* - area where content is introduced and can be displayed in different formats (lists views, record views, etc.)
- *Footer* - static area for information about developer credits

Figure 12 – *Application Layout* definition

By definition, the pattern header has customizations points like site logo, application name, background color or dimension (height and width). It can use other patterns like: an *Application Switch*, *Menu*, *Help Link* or *Personal Information*, as shown in the picture below.

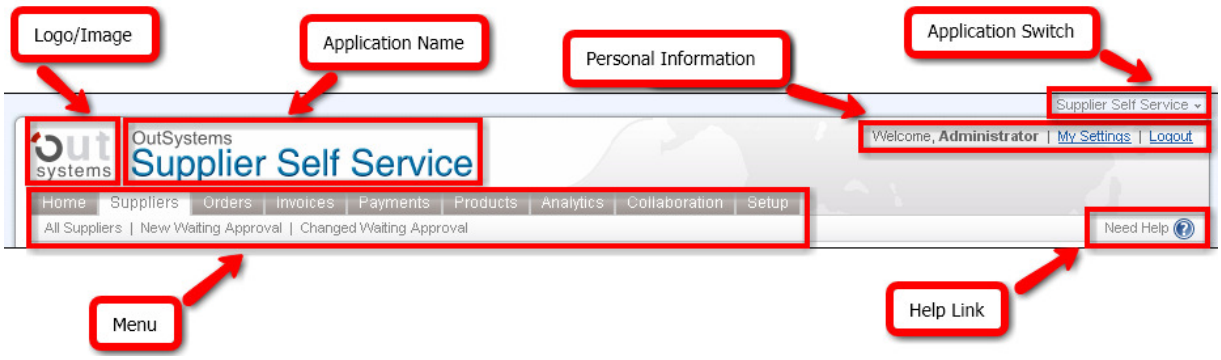


Figure 13 – *Menu* with customization points and children patterns

Our application is very simple, so we won't need all of these patterns. We will only instantiate the *Menu* pattern over the Header pattern. The form of the *Menu* pattern states that we can use the pattern sub-menu (2nd level menu), however we won't use it. When *Menu* is instantiated it doesn't have any Menu Entries, so there isn't any picture to show this instantiation.

Now we are going to implement the management of an entity called "Account". For that purpose, we instantiate the *Core Entity* pattern that receives an entity as a mandatory input. Let the input entity be defined like:

- Entity - Account
- Attributes:
  - account Name : Text
  - phone : Phone-Number
  - address : Description
  - industry : Look-up

This means the entity Account is defined by four attributes: an account name of type *Text*, a phone of type *Phone-Number*, an address of type *description* and an industry of type *look-up*.

Core Entity pattern is hooked by a *Menu Entry* and is composed by other 4 child patterns: *Core Entity Home*, *Show Form*, *Edit Form* and *Create Form*. The instantiation of the *Core Entity* affects the *Menu* pattern, by adding a new entry, and affects the *Page Title* and *Web Page Content* patterns. When we navigate through *Core Entity* children patterns, the respective *Menu Entry* will remain activated, but *Page Title* and *Web Page Content* patterns will change accordingly.

The default content of a *Core Entity Home* is defined by an *Entity List* pattern. *Entity List* receives a record list as input. However, once the *Entity List* was instantiated by *Core Entity Home* and this one by *Core Entity*, the input of the *Entity List* was automatically filled with the record list of the entity provided by the *Core Entity*. Hence, *Entity List*'s input is a record list of accounts.

Page Title is a string with the plural of the name of the entity. The plural is defined at (and is a customization point of) *Core Entity*. There is also a link to navigate to *Create Form* pattern.

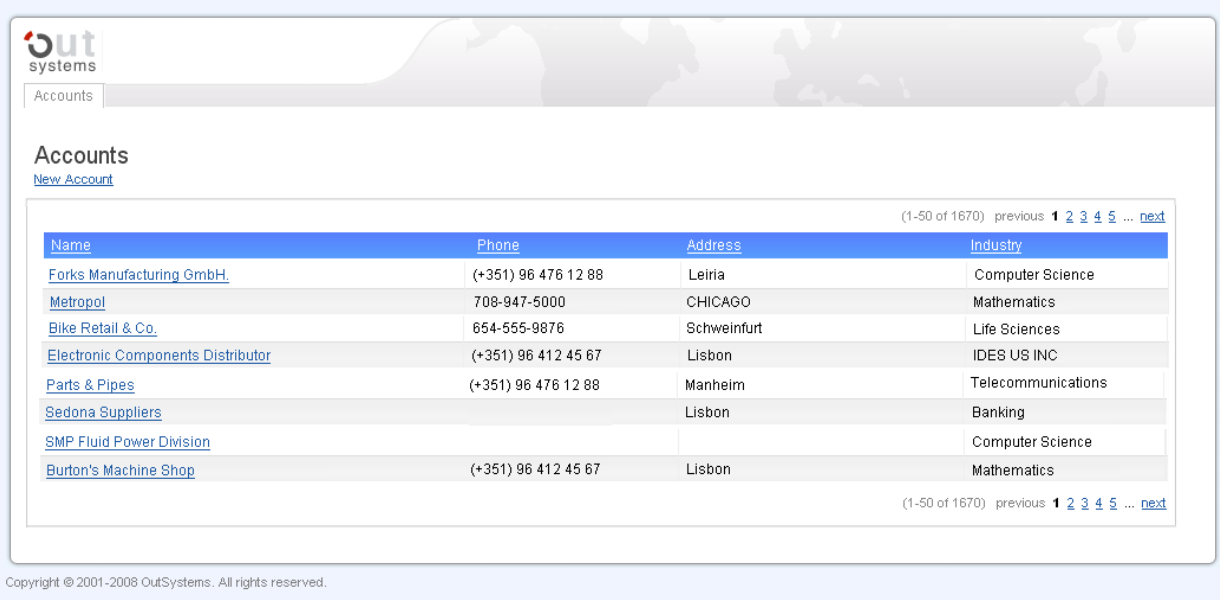


Figure 14 – *Entity List* composed with *Sort By Column*, *Pagination* and *Links to show*

Show Form pattern is composed by *Show Form Field* patterns and receives a record as input. When instantiating a *Show Form*, a set of *Show Fields* are also instantiated for each attribute of the record. Once the *Show Form* was instantiated by *Core Entity,* it was automatically filled in by a record from *Core Entity*. Hence, *Show Form*'s input is an account record.

Show Form is also composed by a *Button Area* pattern. A *Button Area* pattern is a set of actions or operations associated with a *Form*. In the context of a *Show Form*, a button area is materialized (by default) in three buttons/actions: "Edit", "Clone", and "Delete". Customization points of a *Button Area* include adding or removing buttons from the pattern. However, adding custom buttons implies that the developer must implement all the business logic associated with them. *Show Form* may have the same button area above and below the form (customization point).

Page Title is a string beginning with the name of the Entity followed by the name of the record.

Figure 15 – *Show Form* (of the 1st record on the previous entity list)

Edit Form pattern instantiation is similar to *Show Form,* but composed by *Edit Form Field* patterns. The difference is that there are a set of predetermined input boxes for each attribute data type. For example, if the data type of an attribute is Text, the *Edit Field Form* will use an input text. If it is Description, the *Edit Field Form* will use a text area. If it is a Look-up, the Edit Field Form will use a drop-down box.

Edit Form also has a *Button Area* pattern with the buttons: "Save", "Save & New" and "Cancel". *Page Title* is a string beginning with “Edit” followed by the name of the Entity and connected to the main name of the record.

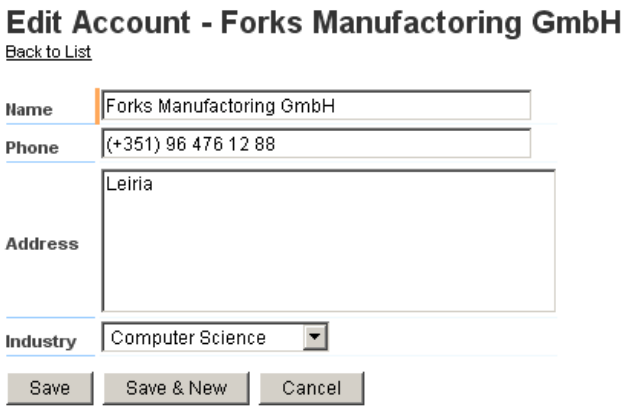


Figure 16 – *Edit Form* (of the first record on the previous *Entity List*)

Create Form pattern is almost the same as *Edit Form* pattern, the difference is that it doesn't receive a record as input. Input is an entity because the *Create Form* pattern needs to know the information about how to generate the various *Edit Form Field* patterns for each attribute. The orange mark means that the field is mandatory.

Create Form has a *Button Area* pattern just like *Edit Form*.

Page Title is the same as *Edit Form*, but the string "New Account" is displayed instead of the name of the record.

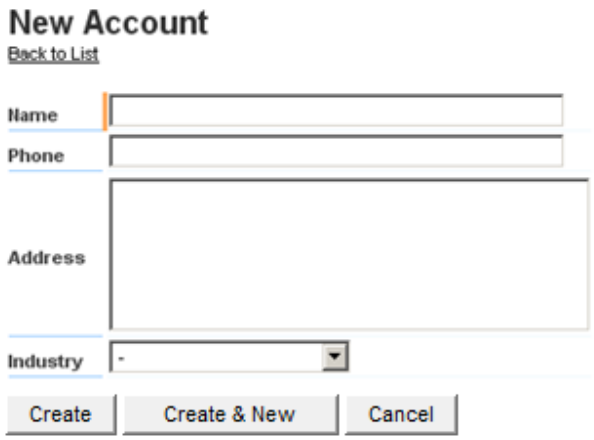


Figure 17 – *Create Form*

We have just implemented the basic management (creation, edition, deletion, and listing) of a core entity Account. The application is ready for deployment; however it is not finished yet. Contact management must also be implemented to finish our initial requirements.

We will develop our Core Entity pattern with the following entity as the input:

- Entity Name: Contact
- Attributes:
    - first_name : Text
    - last_name : Text
    - e-mail : E-mail
    - account : Account Identifier

As we did for entity Account, we will reproduce the same process for entity Contact. The screens generated for *Core Entity* pattern, corresponding to the entity Contact, are similar to the ones from entity Account. Although the instantiation of the patterns is identical, the composition of some patterns is slightly different. Entity Account doesn't have any dependencies with other entities, contrasting with entity Contact. Attribute account in Contact entity corresponds to an Account record. This means *Edit Field Form* (included in patterns *Edit Form* and *Create Form*) corresponds to the attribute of type Account and will be materialized as a drop-down box.

Figure 18 – *Menu* with a new *Menu Entry* (with entity Contacts selected)

Now that we have a relationship between Contacts and Accounts, we can extract more semantic from it. *Master/Detail* is a pattern that the framework may suggest the developer to use. This means that we will have an *Entity List* of Contacts directly linked with the account shown in *Show Form* pattern.

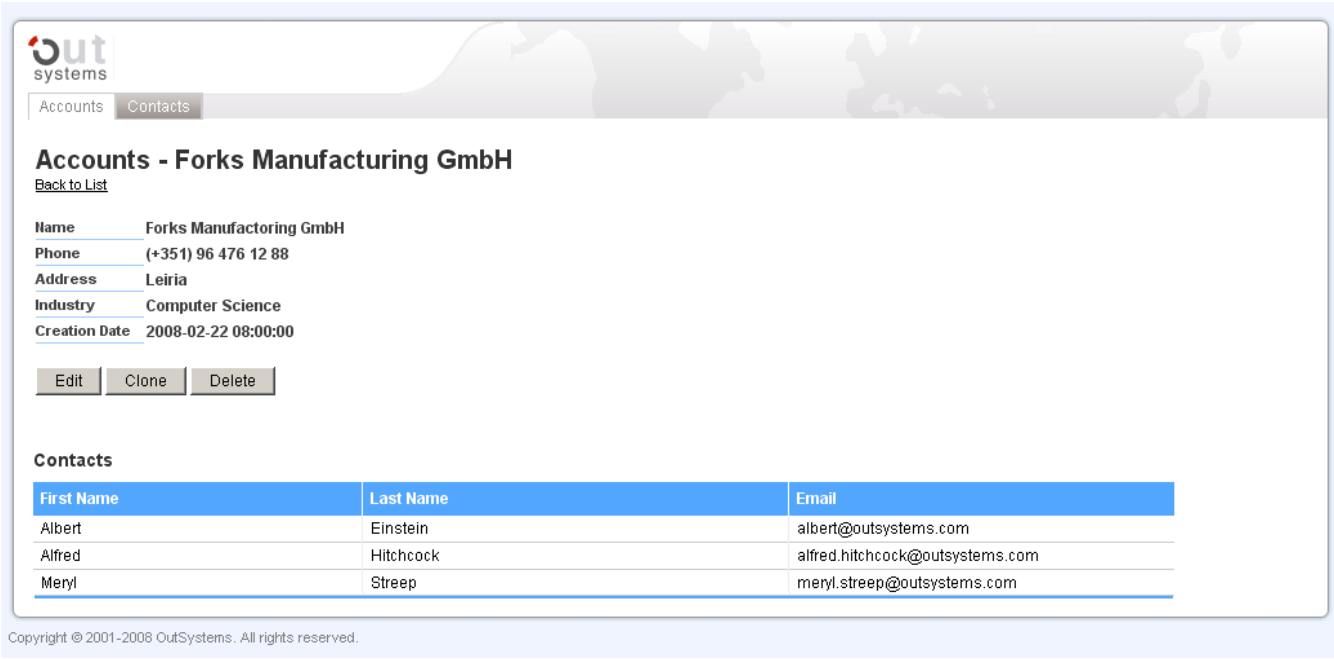


Figure 19 – *Master/Detail* pattern (with one detail list)

Finally we have a web application ready for production, but we can enrich it a little bit more. Imagine that we deal with thousands of records, and we want to search for a certain item. Navigation page by page doesn't seem to be a good solution, so we are going to add some filters to the main *Entity List* of the entity Account.

By definition we know that *Entity List* pattern can use a *Filter* pattern. A *Filter* pattern is composed by a dedicated area above the list a two buttons corresponding to "Apply Filter" and "Reset". The first button applies the filter and retrieves the information filtered. The second applies an empty filter, returning all the items from the table.

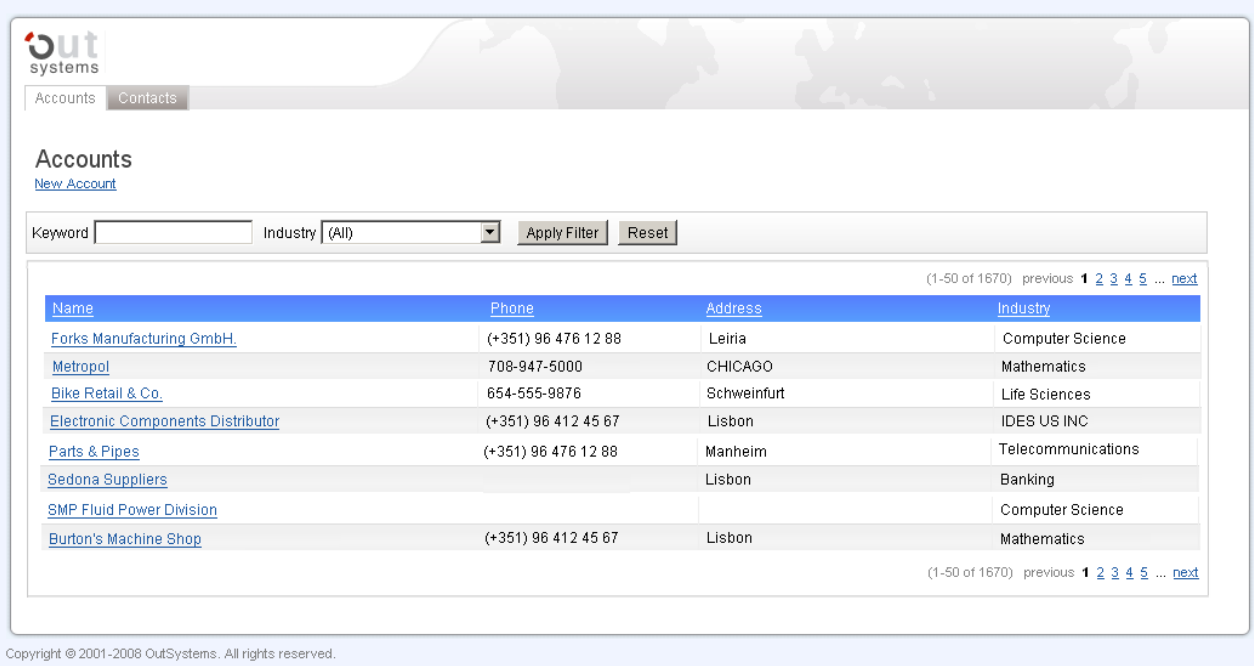


Figure 20 – *Entity List* with *Input Text Filter* and *Drop-down Filter*

The *Filter* pattern, shown in figure 20, uses the two children patterns: *Input Text Filter* Pattern (searches for all the attributes of type *Text*, by default) and *Drop-down Filter* (associated with a Look-up).

Though we added this filter afterwards to exemplify the composition of patterns. This and other patterns can and should be automatically composed by default, allowing the developer to create a full application with minor effort.

7 Conclusion

Pattern-driven web development, based on interaction design patterns, is becoming real as time passes by. In certain contexts, like storage and retrieval, programs are being developed in a repetitive way, allowing us to extract patterns. Composing pattern over pattern, based on pattern-languages for web development, is the way to increase languages' expressiveness, by giving more semantic to patterns. That semantic may be materialized on more functionalities and possible pattern compositions that the framework may suggest the developer. Patterns allow us not just to improve our development processes but are also a good way of documenting knowledge.

The goals were achieved. We produced a mental framework for defining friendly Web 2.0 storage and retrieval applications. A pattern-language was defined together with a set of patterns previously identified in enterprise applications, rich in information storage and retrieval. This language syntax was displayed by a pattern format that specifies built-to-change patterns and supports their composition. Every pattern was validated with stakeholders.

Today, patterns included in these thesis (and much more) are being described at OutSystems. However, as more patterns are added, managing and documenting them is not easy to accomplish. We don't have a global vision over the patterns, there is not a strict relationship between the information about the patterns and the code that implement them, and there is no support for patterns versioning. Though this is not central now, on time it will become necessary to develop a framework to help us cataloguing, documenting, structuring, and managing patterns.

Also, there will always be a problem if we want to do something very specific to a pattern. One solution for this is to explode the pattern and dispose all the internals based on the primitives of the language. This leads to loss of the semantic of the pattern. The second solution is to add that specific characteristic or operation as a customization point. It will always cost trade-of: either we continuously add new customization points that respond to every developer demands but will multiply the number of combinations; or we will let the developer explode the pattern in benefit of a clean pattern with a small and common set of customization points.

In the future, we believe patterns, with all inputs and customization points well defined, are enough to build a full friendly web application, only based on data model. Imagine a world where we design your data model, around core entities, and the whole application is auto-generated. Following this path, we believe we are not too far from that.

7.1 Future Work

In order to completely prove that our study was successful we still need to measure how these patterns, if implemented, enhance productivity. In the future, we need to have statistics on how these patterns contribute to less effort on building user interfaces.

It is also important to come up with solutions for an all-in-one framework to keep patterns' code near patterns' documentation. These frameworks should respond to other requirements such as: supporting management of patterns, creation and maintenance of patterns specification (including videos to represent the interaction through time), analyze impacts of changes in a given pattern, versioning and security. Of course, it is also critical that these patterns are implemented.

It is becoming clear that academic world needs more and more input from the business to continuously analyze other patterns from other stereotypes. Simultaneously, it is essential to validate the outputs with stakeholders from both academic and business.

Connecting patterns was never done before in pattern-driven web development, but we believe next studies after this one will follow a pattern language approach to construct the next big full pattern-language description.

Bibliography

[1] **Coad, Peter.** *Object-oriented patterns.* s.l. : Communications of the ACM, 1992.

[2] **Alexander, C.** *The Timeless Way of Building.* Oxford, UK : Oxford University Press, 1979.

[3] **Alexander, C., Ishikawa, S. and Silverstein, M.** *A Pattern Language: Towns, Buildings, Construction.* New York : Oxford University Press, 1977.

[4] **Gamma, E., et al.** *Design Patterns: Elements of Reusable Object-Oriented Software.* Indianapolis : Addison Wesley Professional, 1995.

[5] **Buschmann, F., et al.** *Pattern-Oriented Software Architecture.* New York, NY : John Wiley & Sons, Inc., 1996.

[6] **Fowler, Martin.** *Analysis Patterns: Reusable Object Models.* Menlo Park : Addison Wasley, 1997.

[7] **Dearden, Andy and Finlay, Janet.** *Pattern Languages in HCI: A critical review.* UK : s.n., 2006.

[8] **Granlund, Åsa, Lafrenière, Daniel and Carr, David A.** *A Pattern-Supported Approach to the User Interface Design Process.* New Orleans, USA : Proceedings of HCI International 2001 - 9th International Conference on Human-Computer Interaction, 2001.

[9] **Mahemoff, Michael J. and Johnston, Lorraine J.** *Principles for a Usability-Oriented Pattern Language.* Australia : Computer Human Interaction Conference, 1998. Proceedings., 1998.

[10] **Fowler, Martin.** Writing Software Patterns. *martinfowler.com.* [Online] Aug 1, 2006. [Cited: Aug 2008, 20.] http://martinfowler.com/articles/writingPatterns.html.

[11] **van Welie, Martijn and van der Veer, Gerrit C.** *Pattern Languages in Interaction Design: Structure and Organization.* The Netherlands : Proceedings of Interact '03, 2003.

[12] **Fincher, S.** *Perspectives on HCI patterns: concepts and tools.* Florida, USA : CHI 2003 Workshop, 2003.

[13] **Fincher, Sally.** *Analysis of Design: An Exploration of Patterns and Pattern Languages for Pedagogy.* UK : Journal of Computers in Mathematics and Science Teaching, 1999.

[14] **Lin, J. and Landay, J. A.** *Damask: A Tool for Early-Stage Design and Prototyping of Multi-Device User Interfaces.* s.l. : 8th International Conference on Distributed Multimedia Systems, 2002.

[15] **May, D. and Taylor, P.** *Knowledge Management with Patterns.* USA : ACM Association for Computing Machinery, 2003.

[16] **Henninger, S.** *An Organizational Learning Method for Applying Usability Guidelines and Patterns.* Germany : Springer-Verlag, 2001.

[17] **Molina, P., Meliá, S. and Pastor, O.** *User Interface Conceptual Patterns.* Germany : Springer-Verlag, 2002.

[18] **Fowler, Martin.** Patterns in Enterprise Software. *martinfowler.com.* [Online] Feb 19, 2005. [Cited: Aug 20, 2008.] http://martinfowler.com/articles/enterprisePatterns.html.

[19] **Fowler, Martin and Rice, David.** *Patterns of Enterprise Application Architecture.* s.l. : Addison-Wesley, 2003.

[20] **Trowbridge, Mancini, Quick, Hohpe, Newkirk, Lavigne.** Enterprise Solution Patterns Using Microsoft .NET. *Microsoft patterns & practices Developer Center.* [Online] Microsoft Corporation, June 2003. [Cited: Aug 20, 2008.] http://msdn.microsoft.com/en-us/library/ms998469.aspx.

[21] **Alur, Deepak, Crupi, John and Malks, Dan.** *Core J2EE Patterns: Best Practices and Design Strategies.* s.l. : Prentice Hall / Sun Microsystems Press, June, 2003.

[22] **Hohpe, G. and Woolf, B.** *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* s.l. : Addison-Wesley Professional, 2004.

[23] **Trowbridge, Roxburgh, Hohpe, Manolescu, Nadhan.** Integration Patterns. *Microsoft patterns & practices Developer Center.* [Online] Microsoft Corporation, June 2004. [Cited: Aug 20, 2008.] http://msdn.microsoft.com/en-us/library/ms978729.aspx.

[24] **Evans, Eric.** *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston : Addison-Wesley, 2004.

[25] **Hay, David.** *Data-Model Patterns: Conventions of Thought.* s.l. : Dorset House, November 1995.

[26] **Yahoo!, UED (User Experience & Design) group.** Yahoo! Developer Network. *Design Pattern Library.* [Online] Yahoo!, 2006. [Cited: Aug 20, 2008.] http://developer.yahoo.com/ypatterns/.

[27] **van Welie, Martijn.** Pattern library. *Welie.com - Patterns in Interaction Design.* [Online] 2007. [Cited: Aug 20, 2008.] http://www.welie.com/patterns/.

[28] **Toxboe, Anders.** User Interface Design Patterns Library. *UI Patterns.* [Online] 2008. [Cited: Aug 20, 2008.] http://ui-patterns.com.

[29] **Erickson, Tom.** The Interaction Design Patterns Page. *Tom Erickson's Homepage.* [Online] 2007. [Cited: Aug 20, 2008.] http://www.visi.com/~snowfall/InteractionPatterns.html.

[30] **Mahemoff, Michael.** Ajax Design Patterns. [Online] 2006. [Cited: Aug 20, 2006.] http://ajaxpatterns.org/.

[31] **Coad, P. and Mayfield, M.** *Workshop Report: Patterns.* s.l. : Addendum to the Proceedings of OOPSLA '92, 1992.

[32] **Gabriel, P.** *Patterns of Software: tales from the software community.* UK : Oxford University Press, 1996.

[33] **Borchers, J. O.** *A Pattern Approach to Interaction Design.* Germany : AI and Society, 2001.

[34] **van Welie, Martijn and Klaasse, Bob.** *Evaluating Museum Websites using Design Patterns.* The Netherlands : Satama, December 2004.

[35] **Rivers, Hagan and Rivers, David.** *Web Apps Tour 2007: Learning from Successful Designs.* North Andover, MA : User Interface Engineering, 2007.

[36] **Yahoo!, UED (User Experience & Design) group.** The Lifecycle of a Pattern. *Design Pattern Library.* [Online] 2006. [Cited: Aug 20, 2008.] http://developer.yahoo.com/ypatterns/page.php?page=lifecycle.

[37] **OutSystems S.A.** About an eSpace. *OutSystems Service Studio 4.1 Help.* [Online] 2006. [Cited: Aug 20, 2008.] http://community.outsystems.com/help/servicestudio/4.1/default.htm#%3E%3Epan=2.

[38] **Ceri, Stefano.** *Designing Data-intensive Web Applications (The Morgan Kaufmann Series in Data Management Systems).* s.l. : Morgan Kaufmann, 2003.

[39] **Korfhage, Robert.** *Information Storage and Retrieval.* s.l. : Wiley, 1997.

[40] *Salesforce.* [Online] [Cited: Aug 2008, 1.] http://www.salesforce.com.

[41] **Sutcliffe, A.** *On the Effective Use and Reuse of HCI Knowledge.* USA : Association for Computing Machinery Press, 2000.

Appendix A – Short List of Identified Patterns (Flatten and Unfiltered)

=Hour Picker	=Mass Mail Campaigns
=Date Picker	=Geographical Location Maps
=Color Picker	=Server Side Control Integration
=Multiple picker with incremental inputs (add more, remove)	=Web-Based Instant Messaging
=Double Post Protection	=External Operations Orchestration with Transactional Control or Compensation Actions
=Blocking Wait while Displaying Feedback to End-User on Submit Request	=SNMP Integration
=Example/Help Text That Disappear On Input Click/Type	=FTP Integration
=Keyboard shortcuts	Data Centric
=Focus control	=Entity Kinds
=Last Used Suggestion on Input	==Core
=Centralized Create New	==Lookup
=Form View / Edit over Entities	===Static
==Save Changes Forms Confirmation Question On Exit / On Submit	===Dynamic
==Button Area	==Generic
==Save with Data Consistency / Conflict Detection	=Relation Kinds
==Show Record With Inline Edition Capabilities	==Specialization / Sub Type / Inheritance (1-1)
==Toggle Show/Edit Record	==Detail (1-N)
=Input Validation While Typing	==Parent (1-N) (Recursive) / Tree
=Form Input Dependencies Validations & Reactions	==Link (N-N)
==Input Validation and Content Change According with End-User Selection on Other Input	===Simple
==Input Enable/Disable Triggered by Changes on Other End-User Input	===With Attributes
==Combo Box with Refreshable Contents Triggered by Changes on Other End-User Input	===Graphs
=Sliders for Numeric Inputs	=Housekeeping Processes (CRUD) on Lookup Entities
==Single Slider for range interval	=Fulfillment Conditions for N-N relations
=Auto Complete While Typing Input	=Create and Maintain Relations Around Core Entities

=Application Help Link	==Core Entity Home
=Help or Other Content On MouseOver	==Core Entity List
=Help or Other Content On Click	==Core Entity Detail (Master/Detail)
=Help or Other Content On Text Selection	==Quick Create
=Help or Other Content On Input Click/Selection	==Details Relation - Create and Associate Detail Entity to Master Entity
=Help for this Page	===Details Relation - Model and UI Support with Common Operations (Add, Remove, Move)
=Contextualized Help for this Page	===Attach Documents to Entity
=Runtime Customizable, Positional and Resizable Blocks / Portlets	===Attach Comments to Entity
=Runtime Sortable, Resizable and Editable Tables	===Create and Maintain (1-N) Header Entity / Line Entity
=Data Set / Data Block (discard)	====Create and Maintain (1-N) Header Entity / Line Entity With Status
=HTML Frames (discard)	==Details/Specialization Relation - Collapse Related Entities Into Core Entity for Creation and Maintenance
=Preview Link Destination On MouseOver	==Parent Relation - Define Entity as an Assembly of Entities of the Same Type
=Drag & Drop Containers Into Grouped Lists Records or Maps	==Parent Relation - Model and UI Support with Common Operations (Add, Prune, Move, Set Parent)
=Move Items Between Two Lists Records Using Drag-and-Drop	==Specialization Relation - Entity Inheritance / Sub Type
=Order Elements of Table Records Using Buttons	==Specialization Relation - Create and Maintain (1-1) Mandatory Entity Master / Entity Extension
=Order Elements of Table Records by Drag and Drop	==Specialization Relation - Create and Maintain (1-0/1) Optional Entity Master / Entity Extension
=Combo Box With Editable Input	==Link Relation - Network & Graphs Model and UI Support with Common Operations (Add, Remove, Move, Link, Unlink)
=List Records With Inline Edition Capabilities	=Look Up Entities Seed Data
==List Records With Inline Input Edition	==Static LookUp
==List Records With Inline Show/Edit Toggle	==Dynamic Lookup
=Server-Side Update When Leaving/Changing the Input Field	=Audit Data Changes
=Expand / Collapse Area	==Aggregate Audit for Related Entities Data Changes in Master Entity
==Animated Expand / Collapse Area	==Audit Data Change History Search by Who, When and Why
=List without pagination and feeding on demand using a scrollbar	==Time-Travel Over Entity History
=Selection / Filter in Different Page of Results Listing Page (E.g.	==Data Changes Secured Observation (Read-Only) Backoffice

Google)	with Filters
=List Records with Filters	==Aggregated Data Changes Reports by Time, Showing Final State, and Evidencing Changed Attributes
==List Records with Filters that accept list of values	==Aggregated Data Counts Evolution in Time
=Entity List View / List over Entity	=End-Users Subscriptions & Change Notifications
==List Records with a New button	=End-Users Update Reminders
==List Records with Actions Column	=See Also...
= Detail Entity List View / List of Detail Entity	==See Also... Based on Access Frequency
== Detail List Records with New buttons	==See Also... Configurable in Runtime
=Alphabetical Index Filter on Sorted List Records	=Top 10
=Container Refresh Upon End-User Action	==Top 10 Most Accessed
=List Records with Links/Buttons/Images to Update Record and Refresh	==Top 10 Last Created (this is not Top 10, it's Recent Items List)
==Refresh / Link to Self screen	=Count Entity Access/Changed Times
=Auto-Timer Refresh of Screen	=Decorate Entities with Tag / Label Mechanisms
=Status Monitoring Refresh of Screen	=Text Automatic Classification into Buckets
==Refresh Part of Screen on Auto-Timer	=Decorate Entities with Discussion Mini-Forums
==Refresh Part of Screen on Data Change	=Search
==Refresh Part of Screen on Operating System Process	==Search Over Multiple Applications
==Refresh Part of Screen on Business Process Change	==Search Over Single Applications
=Progress Bar	==Search Over Entities Set
=Process Feedback Monitoring Upon End-User Action	==Search Over Screen Content
=Non-Intrusive Content Rating Mechanism	==Search Over Documents and similar resources
=Timeline Widget	==Text Search Inside Selected Entity Attributes
=Popup Alerts and Reminders	==Advanced Search Syntax
=Submit Feedback	==Skip Search result list screen when there is only one hit
=Send to a friend	=Text Similarity Functions
Data Navigation & Presentation	=Text Differences Functions & Display
=Tag Cloud in Content Heavy Applications	=Clone Entity Record
=Image Maps	=Convert Entity Record into another Entity Type Record
=Favorites	=Data Backup and Restore
=Send to Printer	==Archive Data for Later View
=Web Applications Layout	==Snapshot Data
==Preview and Select Layout Template in Design Time	==Staging and Data Migration

==Preview and Select Layout Template in Runtime	==Snapshot Data Restore / View
==Design Layout Templates	=Binary Data Storage
==Styles Modularity	=Efficient Bulk Maintenance of Large Data Sets
==Layout control without changing Stylesheet	==Efficient Bulk Create and Update of Large Data Sets
==Page Title	==Efficient Bulk Delete of Large Data Sets
=User Defined Views	==Efficient Bulk Set Operations
=External Portal Integration (Layout, Menus, Header and Footer)	=Secondary Keys
=Site Maps	=Partial Entity Update
=Navigation Menu	=Cascade Delete With Condition
==Menu Item	=Data Partitioning
==Menu Sub-Item	=Database Split
==Standalone Navigation Menu	=Database Views
==Navigation Menu Integrated with Other Applications	=Data Quality Reports with acknowledge
==End-User Customizable Menus	=Data Cache
==End User Customizable Pages	==Server Side Data Cache
=Where am In Application Hierarchy Path & Menu Level...	==Client Side Data Cache
==Back To Last Filter	Process Centric
==Back to top	=Ad-Hoc State Machine
=Where Have I Passed to Reach Here / Navigation History	=Inboxes with Monitoring and Alerts
=Back navigation	==Inboxes with shared tasks (and claim mechanisms)
=Links to Navigate to Edit Screens	==Manage Ticked Type of Entity With Creation / Monitoring & Change Notifications / History
=Paginate List Records	=Runtime Configurable & Monitorable Process Modeling (Work Flow)
==Paginate List Records With Pages Links	==Role-Based Tasks Transition Processes
==Keep Columns Headings Visible when Paginating	==Estimated time to complete
=Set Number of Table Record Lines in Runtime	==Process Monitoring (Queues, Status Diagram, ...)
=List Records With Expandable Groups and Sub-Groups	==Activity Definition by End-User
=List Records with Order By Columns	=Design Time Process Modeling (Work Flow)
=Persist List Records Filter Through Sessions	==Activity Definition
=List Records with on-over highlight for each line	==Expose Each Activity as a form tab
=List Records with row color schema	=="Save and Proceed" / "Cancel" activity
=Save and Name List Records Filters to Reuse Later	=More Workflow Patterns

== Expire All Passwords	=User Centric
== Set Expiration Time Limit (30 days,. 6 months, 1 year, never...)	=Information Hub
== Set Enforce Password History	Ariba
== Set Minimum Password Length	=Data lock that lasts multiple screens and requests
== Set Password Complexity Requirement	=Personal workspace
== Set Password Question Requirement	=Create shortcut for this page
== Set Maximum Invalid Login Attempts and Lockout Period	=Error report for end user show the error number in the database
=Roles / Profiles Backoffice	=Grid / Form view / Set as default view
=Permission Delegation to another user or group	=Balloon validation message
=Web Services Security	=Define entity as test entity (data not relevant for production)
=Data Access Control for CRUD Patterns	=Questionary
=Support HTTPS	=Grouped tables with expand/collapse for each group
=Require HTTPS	=(i) Information "bug" appears popup div
=Session Settings / Configuration	=Customizable home page (with multiple portlets)
== Set Session Timeout	=Non inline feedback pattern (on top, fixed position)
==Lock sessions to the IP address from which they originated.	=Picker with create new
==Enable caching and autocomplete on login page	=Saved filters with tabbed
==New IP Login Notification	=Advanced picker in editable combo + suggestions
==Setup Audit Trail	=Communication Templates
==Compliance BCC Email	==Set up your Letterhead to standardize the look and feel of HTML email templates.
==Show All Remote Sites	==Management of Email Templates
Internationalization	=Application Setup
=Multilingual in Design Time	==Help Settings
==User Specific Locale	==Home Page Components Management
==Check Spelling	==Manage Home Page Layouts
=Multilingual Lookups	==Tracking HTML Email
=Multilingual Data	=Security
	==Activate this computer

Table 13 – Short List of Identified Patterns in the very beginning of this study

Appendix B – Application Layout Pattern

The next image shows another composition of the *Layout* Pattern.

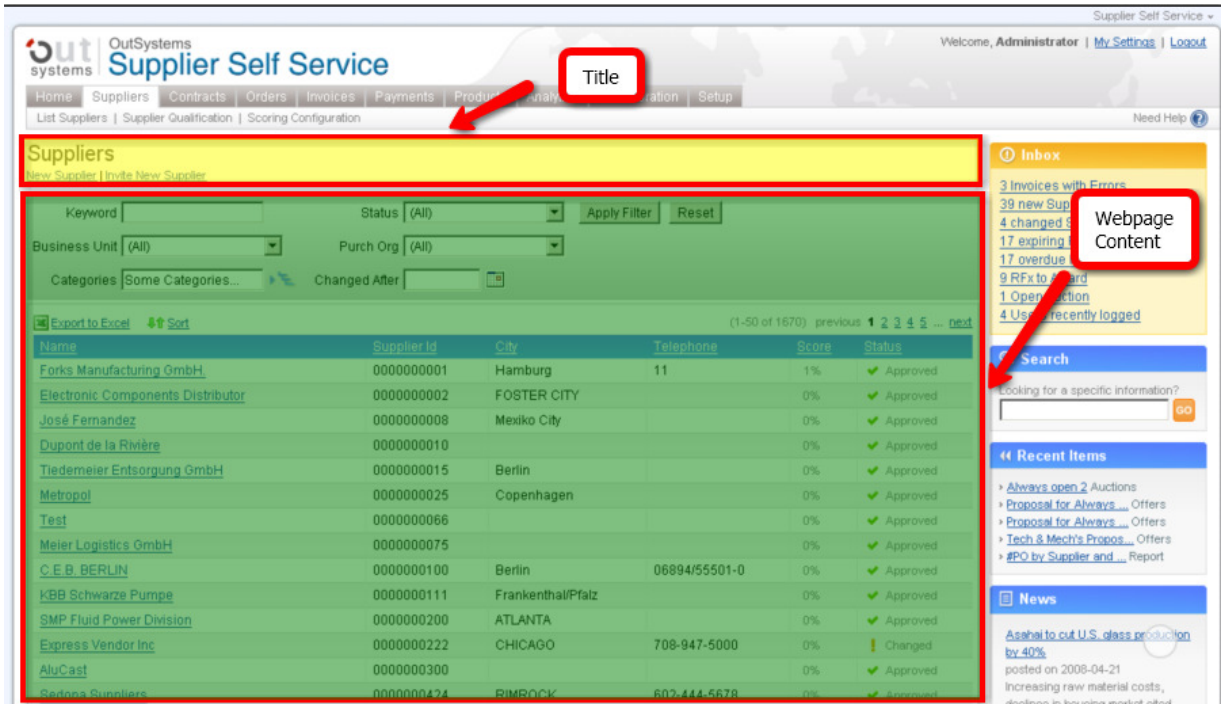


Figure 21 – Application Layout 2 of 2

Appendix C – *Entity List* Pattern

The following figures describe *Entity List* patterns composed with different patterns.

Figure 22 – An *Entity List* composed by several other patterns (Action Column, Link To, Alphabetic Pagination, Saved Filters and List Operations) from Salesforce.com

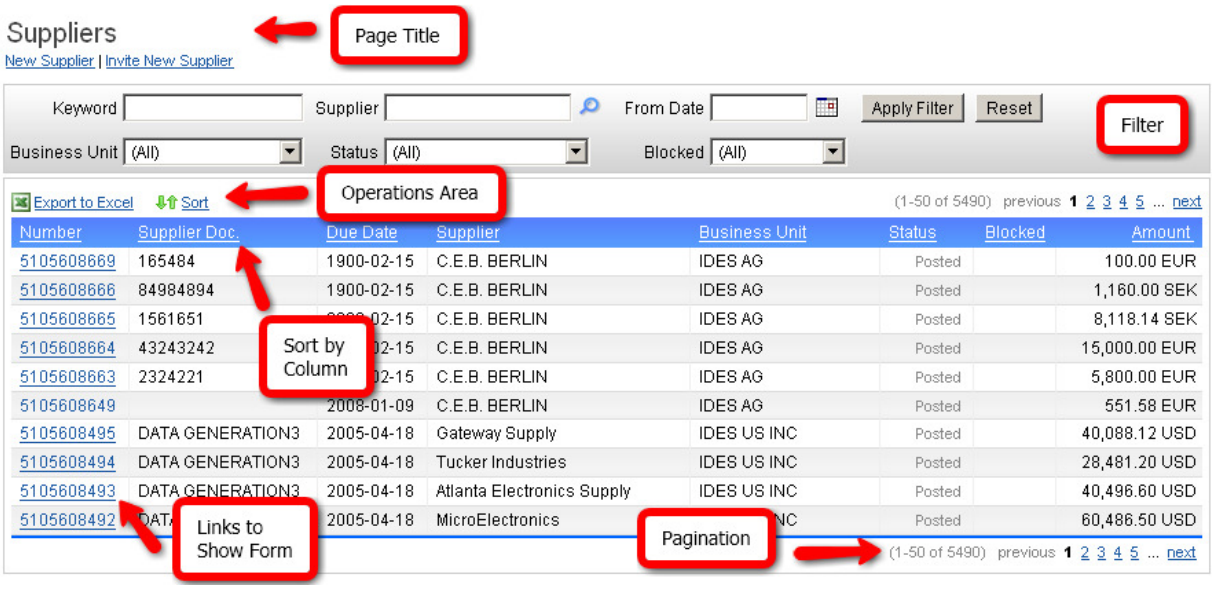


Figure 23 – Entity List from Supplier Self Service

Appendix D – *Master-Detail* Pattern

The following pattern describes a *Master-Detail* pattern composed with a *Show Form* and an *Entity List*.

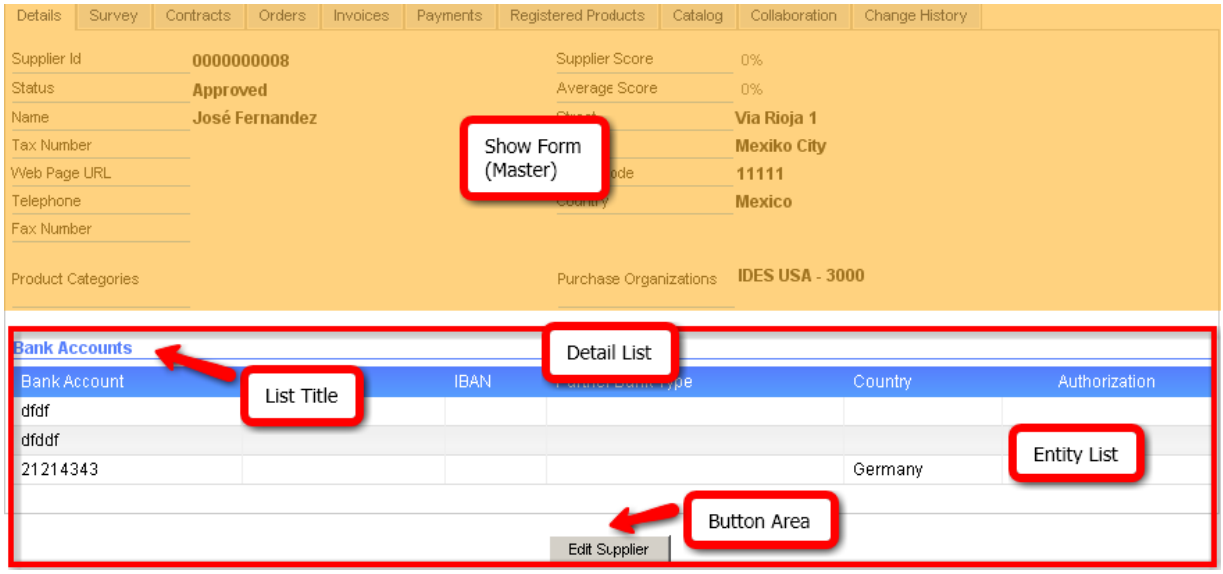


Figure 24 – *Master-Detail* pattern in Supplier Self Service

Appendix E – *Edit Form* Pattern

The following figures present *Edit Forms* from different sources.

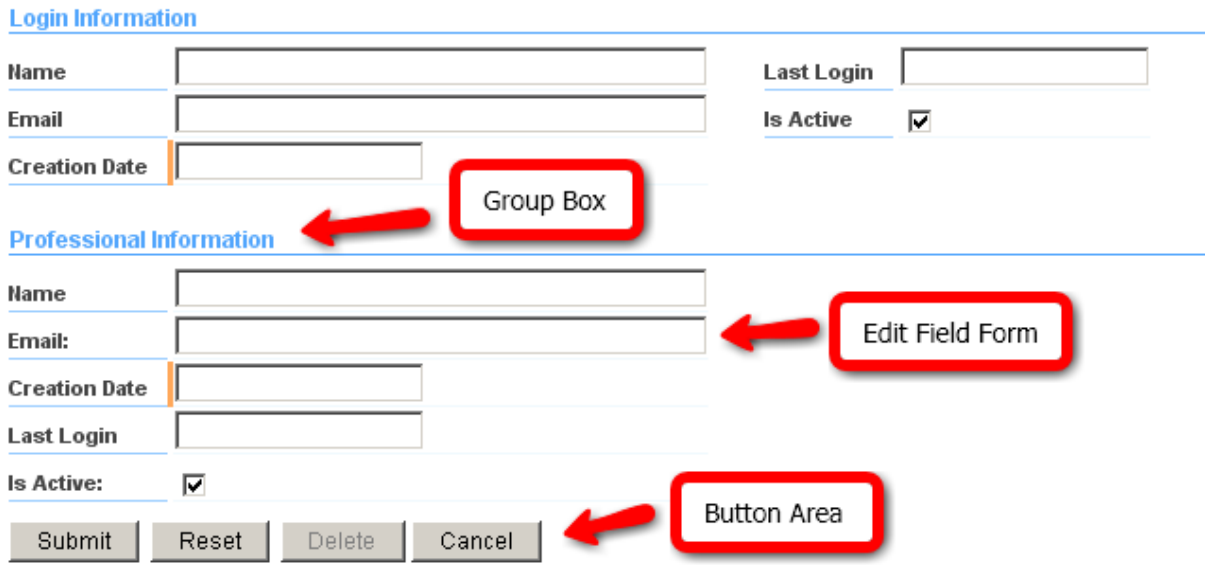


Figure 25 – *Edit Form* pattern from Supplier Self Service

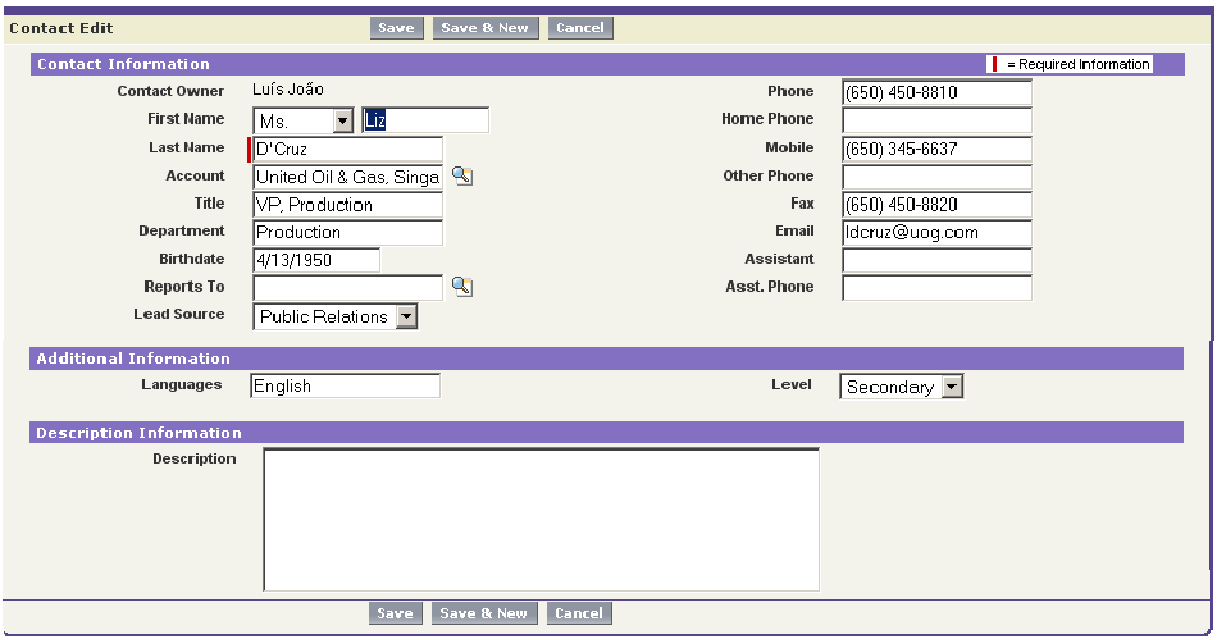


Figure 26 – *Edit Form* pattern (2/2) from Salesforce.com

Appendix F – *Edit Field Form* (EFF) Pattern

The following figures present *Edit Fields Forms* composed by different patterns.

Figure 27 – EFF with drop-down

Figure 28 – EFF with text-area

Figure 29 – EFF with Pop-up Picker

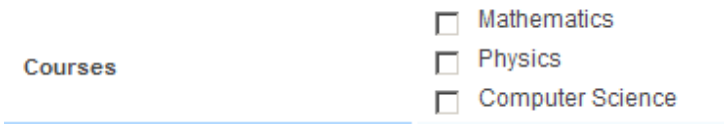


Figure 30 – EFF with Check Boxes

Figure 31 – EFF with Radio Buttons

Appendix G – *Edit Field Form* (EFF) Pattern

The following figures presents *Edit Field Form* pattern from various sources composed with different patterns.

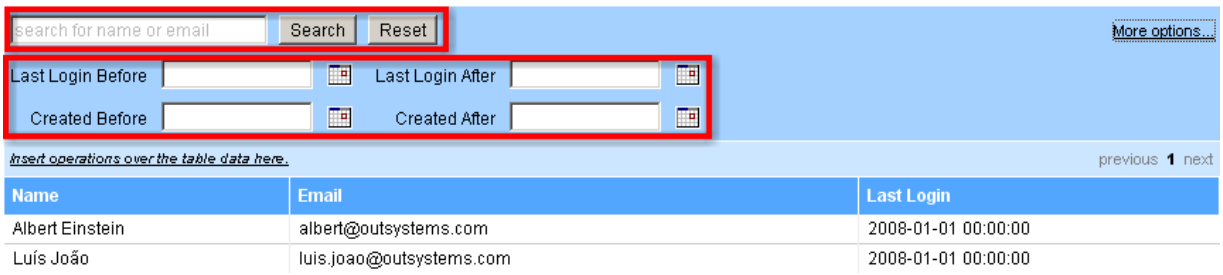


Figure 32 – Filter Pattern from OutSystems' Style Guide

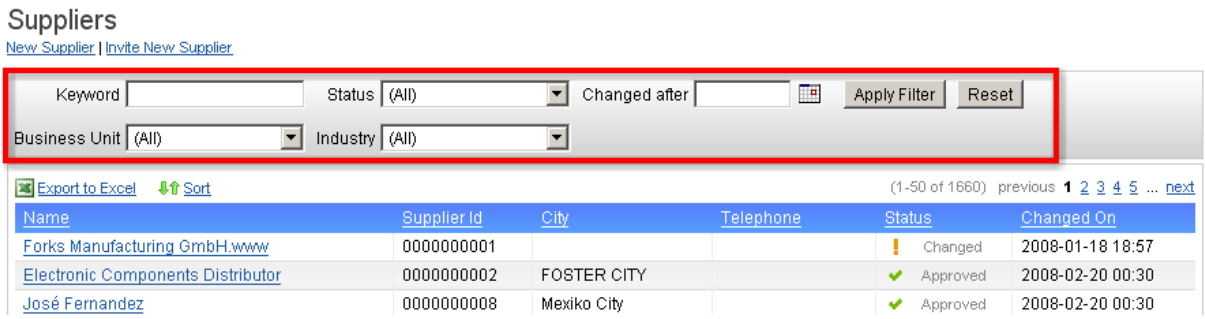


Figure 33 – Filter Pattern from Supplier Self Service (1/2)

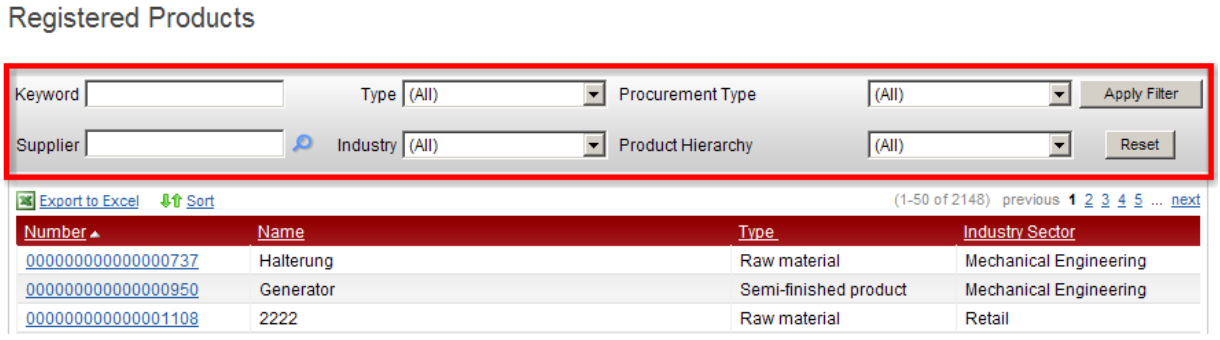


Figure 34 – Filter Pattern from Supplier Self Service (2/2)

Appendix H – *Action Feedback* Pattern

The following figures present Action Feedback patterns from different sources.

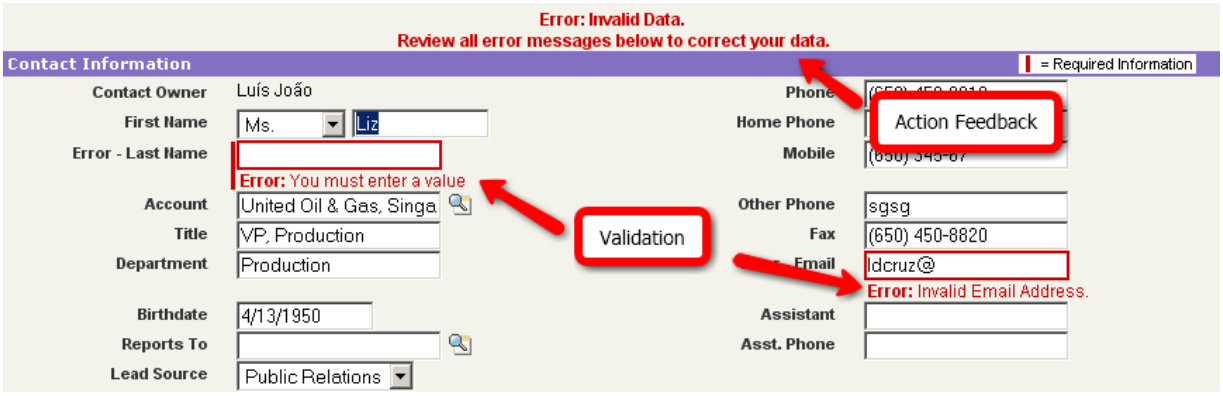


Figure 35 – Action Feedback in Salesforce.com

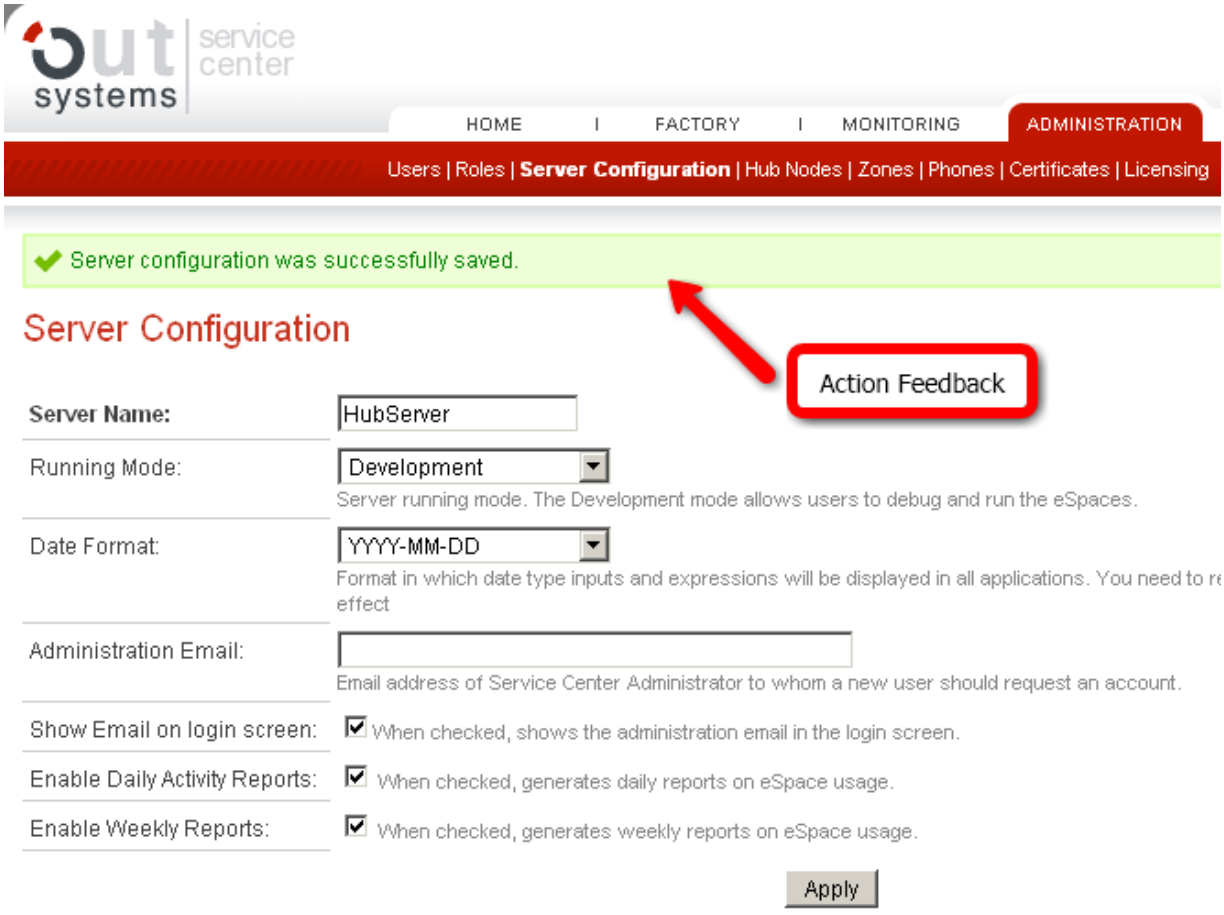


Figure 36 – Action Feedback in Service Studio

Appendix I – *Button Area* Pattern

The following figures present Button Areas in different contexts and different sources.

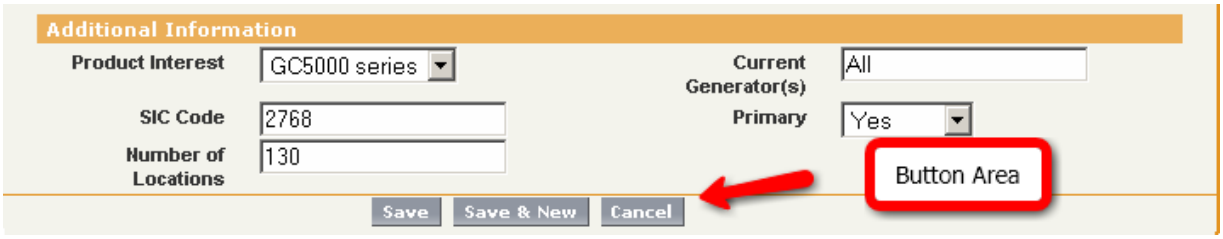


Figure 37 – Button Area (part of *Edit Form* pattern) in Salesforce.com

Figure 38 – Button Area (part of *Filter* pattern) in Supplier Self Service

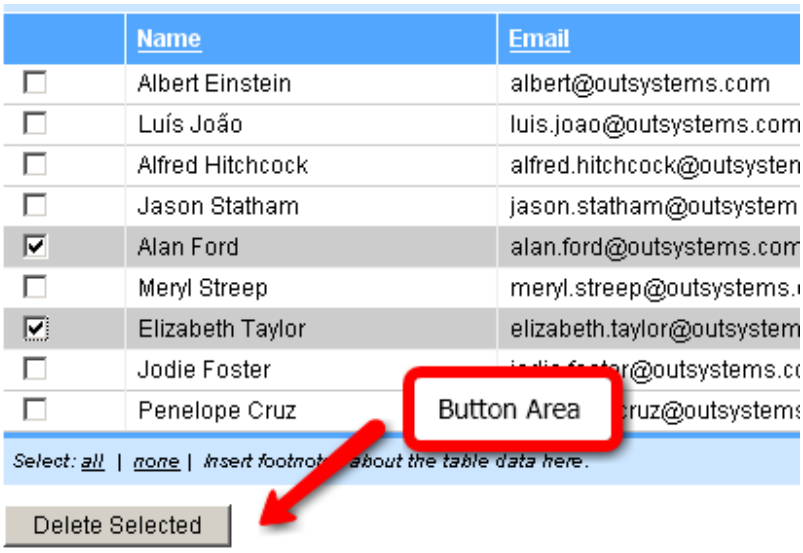


Figure 39 – Button Area (part of *List Operations* pattern) in OutSystems Style Guide

The following figure is not a pattern, but a customization point of a Button Area.

Figure 40 – Customization Point of Button inactivation